TEKNILLINEN KORKAKOULU
Sähkötekniikan osasto

Antti Louko

Tiedostojärjestelmät Mach-ympäristössä

Diplomityö, joka on jätetty opinnäytteenä tarkastettavaksi diplomi-insinöörin tutkintoa varten Espoossa 17. elokuuta 1992.

Työn valvoja Heikki Saikkonen

Työn ohjaaja Timo Sirkiä

TEKNILLINEN KORKEAKOULU DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Antti Louko	
Työn nimi:	Tiedostojärjestelmät Mach-ympäristössä	
Päivämäärä:	17. elokuuta 1992	Sivumäärä: 75
Osasto:	Sähkötekniikan osasto	
Professuuri:	Tik-76 Tietojenkäsittelyoppi	
Työn valvoja:	Professori Heikki Saikkonen	
Työn ohjaaja:	DI Timo Sirkiä	

Tiedostojärjestelmä on oleellinen osa nykyaikaista käyttöjärjestelmää. Sen avulla ohjelmat voivat tallentaa tietoa myöhempää käyttöä varten sekä välittää tietoa muille ohjelmille.

Diplomityön aluksi esitellään lyhyesti tiedostojärjestelmien kehitystä käyttöjärjestelmien mukana. UNIX-käyttöjärjestelmän ja sen tiedostojärjestelmätoteutuksen ominaisuuksia ja ongelmia käsitellään tarkemmin.

Tiedostojärjestelmien ominaisuuksia tarkastellaan ensin yhden koneen ympäristössä ja sen jälkeen tarkastelu laajennetaan hajautettuihin järjestelmiin. Yleisimpien verkkotiedostojärjestelmien ominaisuuksia esitellään ja niitä vertaillaan.

Mach-käyttöjärjestelmän historia selostetaan lyhyesti ja sen ominaisuudet kuvataan pääpiirteissään. Machin ominaisuuksia hyödyntävä yksinkertainen asiakas-palvelinohjelma esitellään pohjaksi samalla periaatteella toimivalle Machin ominaisuuksia käyttävälle tiedostojärjestelmäpalvelimelle.

Seuraavaksi esitellään Machiin tehty käyttöjärjestelmien testaukseen sopiva ohjelmisto, jossa on mahdollista ajaa useita eri käyttöjärjestelmiä saman Mach-ytimen päällä siten että käyttöjärjestelmät käyttävät toistensa tiedostojärjestelmiä Machin kommunikaatio-ominaisuuksia hyödyntäen.

Avainsanat: Mach, käyttöjärjestelmät, tiedostojärjestelmät

HELSINKI UNIVERSITY OF TECHNOLOGY ABSTRACT OF THE MASTER'S THESIS

Author:	Antti Louko	
Name of the thesis:	File systems in Mach environment	
Date:	August 17, 1992	Number of pages: 75
Faculty:	Electrical Engineering	
Professorship:	Tik-76 Information Processing Science	
Supervisor:	Professor Heikki Saikkonen	
Instructor:	M. Sc. Tech. Timo Sirkiä	

File system is an integral part of a modern operating system. Programs use a file system to store data for later use by users and other programs.

This thesis describes shortly the history of development of file systems in parallel with operating systems. UNIX operating system, its file system and its problems are described in more detail.

Features of file systems are presented first in non-distributed systems and then in distributed systems. Some widely used network file systems are presented, evaluated and compared.

The history of Mach operating system is briefly discussed and its most important features are described. A client-server program utilizing primitives of Mach is built as a foundation for a more general file system server program.

Finally, an operating system test environment is presented. The environment makes it possible to run different operating system servers on top of a single Mach kernel. The environment coordinates how servers can access services offered by other servers.

Keywords: Mach, operating systems, file systems

Alkulause

Diplomityöni valvojana toimi Heikki Saikkonen. Työn aihe muotoutui samalla kun opintojeni edetessä tutustuin kiinnostaviksi kokemiini tietojenkäsittelyn alueisiin, kuten systeemiohjelmointiin, käyttöjärjestelmiin, joista erityisesti UNIXiin, tiedostojärjestelmiin, tietoliikenteeseen ja tietoturvaan. Kun Mach tuli saataville, se hankittiin TKK:lle ja sen edut havaittiin, oli mielekkään aiheen valinta helppo.

Ohjaajana toimi esimieheni Timo Sirkiä TKK:n Laskentakeskuksesta. Hänelle ja muille työtovereilleni Laskentakeskuksessa esitän kiitokset siitä, että olen voinut keskittyä diplomityön tekemiseen muiden työtehtävieni häiritsemättä. UNIXin parissa työskenteleviä työ- ja opiskelutovereitani kiitän väsymättömästä ja peräänantamattomasta kritiikistä, joka on epäilemättä parantanut merkittävästi työni ja etenkin sen pohjana olevien ohjelmatuotosten laatua.

Carnegie Mellon University ansaitsee kiitokset siitä, että Mach on vapaasti kaikkien halukkaiden saatavilla. Myös kaikki muut vapaata ohjelmointia tukevat henkilöt ja organisaatiot voivat vastaanottaa samanarvoiset kiitokset.

Espoossa 17. elokuuta 1992

Antti Louko

Sisältö

1 Johdanto 1

2 Käsitteet suomeksi ja englanniksi 2

3 Tiedostojärjestelmien historiaa 4

3.1 TOPS-20 4

3.2 VMS 5

3.3 MS-DOS 6

4 UNIX-käyttöjärjestelmä 7

4.1 UNIXin tiedostojärjestelmän tasot 7

4.2 UNIX-semantiikka 9

4.3 UNIXin ongelmia ja puutteita 10

4.3.1 Yhtenäinen nimiavaruus 10

4.3.2 Nimiavaruuden merkitys 10

4.3.3 UNIXin piilotettu semantiikka 11

4.3.4 Monen kirjoittajan ongelma 12

4.3.5 Epäjohdonmukaisuus 13

4.3.6 Tietokanta- ja lokitiedostot 13

5 Tiedostojärjestelmien ominaisuuksista 14

5.1 Tiedostotyypit 15

5.2 Hakemistorakenne ja tiedostojen nimeäminen 15

6 Hajautetut tiedostojärjestelmät 17

6.1 Läpinäkyvyys 17

6.2 Tiedostojen jakaminen 18

6.3 Puskurointi 18

6.4 Sessiosemantiikka 20

6.5 Tilallisuus ja tilattomuus 20

6.6 Tiedostojen toisto (replication) 21

6.7 Esimerkkejä verkkotiedostojärjestelmistä 21

6.7.1 Sun NFS 21

6.7.2 Berkeley NFS 22

6.7.3 System V RFS 23

6.7.4 CMU RFS 23

6.7.5 AFS 23

6.7.6 Amoeba 24
6.7.7 Sprite 25

7 Mach **28**
7.1 Machin historiaa 28
7.2 Mach filosofiana 29
7.3 Machin perusosat 29
7.3.1 Kehykset ja säikeet 29
7.3.2 Muistin hallinta 30
7.3.3 Kommunikaatio 30
7.3.4 Porttijoukot 32
7.3.5 Eri osa-alueiden suhde toisiinsa 33
7.4 Miten Mach eroaa tavallisista käyttöjärjestelmistä? 33
7.5 MIG 34

8 Yksinkertainen asiakas ja palvelin Machissa **36**
8.1 Määrittelytiedostot 36
8.2 Palvelinohjelma 38
8.3 Asiakasohjelma 39
8.4 Apurutiinit 41

9 Machin ominaisuuksia hyödyntävä tiedostojärjestelmä **45**
9.1 Tiedostojärjestelmän kuvaaminen MIG-kutsuilla 45

10 MIGiin perustuva tiedostojärjestelmäliityntä UNIXiin **51**
10.1 UNIX-palvelin Machin päällä 54
10.2 UNIX-palvelin toisen UNIX-palvelimen päällä 54
10.3 Porttitiedostojärjestelmä 55

11 Johtopäätökset **58**

A VFS-kutsut **60**

B `tserver.c` **64**

C `sym.c` **72**

1 Johdanto

Tiedostojärjestelmä muodostaa olennaisen osan nykyaikaisesta käyttöjärjestelmästä ja sen liitynnästä sovellusohjelmaan. Monista muista käyttöjärjestelmän osista poiketen tiedostojärjestelmän ominaisuudet ja toteutus näkyvät yleensä myös sovellusten käyttäjälle asti.

Tässä diplomityössä tarkastellaan tiedostojärjestelmien kehittymistä aikojen kuluessa. Sekä perinteisten että hajautettujen tiedostojärjestelmien ominaisuuksia tarkastellaan sekä yleisesti että käyttäen UNIXia esimerkkinä. Yleisimpien verkkotiedostojärjestelmien ominaisuuksia vertaillaan. Erityistä huomiota kiinnitetään esiteltyjen toteutusten ongelmiin ja heikkouksiin.

Mach on Carnegie Mellon Universityssä kehitetty käyttöjärjestelmäydin, jossa on tehokkaat ja kattavat peruspalvelut prosessien väliseen kommunikointiin sekä muistin hallintaan. Machia voidaan pitää oliokeskeisenä järjestelmänä ja se tarjoaa myös mekanismit tietoturvan takaamiseksi.

Machin käyttöä esitellään tekemällä yksinkertainen oliopohjainen palvelinohjelma, jonka palveluja käyttäen useat asiakasohjelmat voivat käyttää yhteisiä olioita. Seuraavaksi esitellään esimerkin periaatteita noudattaen tehty käyttöjärjestelmien testausympäristö, jossa saman Mach-ytimen päällä voidaan ajaa samanaikaisesti eri käyttöjärjestelmiä, jotka käyttävät toistensa palveluja, kuten tiedostojärjestelmiä, Machin tarjoamien palvelujen avulla.

2 Käsitteet suomeksi ja englanniksi

Jatkossa esiintyy monia käsitteitä, joille ei ole olemassa yleisesti käytettyjä suomennoksia. Suomennoksia, joiden kohdalla on maininta "tässä", on käytetty kyseisessä merkityksessä vain tässä diplomityössä, ja niiden merkitys muualla on erilainen tai laajempi.

oikeuslista (Access Control List) tarkoittaa tiedostoihin, hakemistoihin ja muihin käyttöjärjestelmien olioihin liittyviä määrittelyjä, jotka kertovat kuka saa tehdä mitäkin kyseisille olioille.

tavu (byte) on yleensä 8-bitin tietoalkio. Joissakin koneissa tavulla tarkoitetaan 7- tai 9-bitin tietoalkiota.

oktetti (octet) on 8-bitin tietoalkio. Nimitystä käytetään tavun asemesta sekaannusten välttämiseksi.

prosessien välinen kommunikaatio (Inter Process Communication) Useimmat nykyiset käyttöjärjestelmät joko pohjautuvat prosessien väliseen kommunikaatioon tai ainakin tukevat sitä tehokkaasti. Jatkossa käytetään myös lyhennettä IPC.

puskurointi (caching) tarkoittaa tässä tiedostojärjestelmän tietojen säilyttämistä lähellä käyttöpaikkaa.

kätkömuistilla (cache) tarkoitetaan erityisesti muistia, jolla nopeutetaan muistin käyttöä pitämällä ajankohtaista tietoa sellaisessa muistissa, johon on nopeampi pääsy.

kehys (task) tarkoittaa Machin yhteydessä prosessin resursseja muistiavaruuksineen ja portteineen. Jos käsitettä haluaa käyttää muualla ja sekaantumisen vaara on olemassa, voi puhua Machin kehyksestä.

säie (thread) tarkoittaa kehyksen tarjoamassa ympäristössä ohjelmaa suorittavaa osaa, jota voidaan luonnehtia virtuaaliprosessoriksi.

kova tiedosto (immutable file) on tiedosto, jonka sisältö ei sen luomisen jälkeen voi enää muuttua. Suora käännös olisi muuttumaton tiedosto, mutta tämä käsite ei ilmaise sitä alkuperäisen käsitteen piirrettä, että tiedostoa ei *voi* muuttaa.

kiinnittää (mount) tarkoittaa UNIXissa jonkin tiedostojärjestelmän sijoittamista hakemistohierarkiaan. Tiedostojärjestelmä voi olla esimerkiksi levypartitiolla sijaitseva tai verkon kautta käytettävä.

monistus (replication) tarkoittaa saman tiedon säilyttämistä kahdessa tai useammassa paikassa joko tiedon häviämisen estämiseksi tai toiminnan turvaamiseksi esimerkiksi verkon virhetilanteissa.

kaatua (crash) tarkoittaa joko itse tietokoneen tai sen käyttöjärjestelmän virhetilannetta, josta ei voida toipua. Tällöin kone ja sen käyttöjärjestelmä joudutaan yleensä käynnistämään uudelleen.

sitoutuminen (commit) tarkoittaa jonkin toimenpiteen tekemistä lopulliseksi. Kun kovasta tiedostosta otettu paikallinen kopio talletetaan muutettuna entisen tilalle atomisena operaationa puhutaan tiedoston sitoutumisesta.

etäkutsu (remote procedure call) tarkoittaa aliohjelmakutsua, joka tapahtuu käyttäen kutsun välitykseen jotain prosessien välistä kommunikaatiota tai tietoverkkoa.

asiakas, palvelin (client, server) Asiakas-palvelin-arkkitehtuurilla tarkoitetaan ratkaisua, jossa yksi tai useampi asiakasohjelma tai -prosessi käyttää palvelinohjelman -tai prosessin palveluja esimerkiksi etäkutsuja käyttäen.

Kannettavuudella (portability) tarkoitetaan sitä, kuinka helppoa jonkin ohjelmiston, kuten käyttöjärjestelmän sovittaminen on jollekin uudelle koneelle.

3 Tiedostojärjestelmien historiaa

Nykyisin tuntemissamme käyttöjärjestelmissä on aina tiedostojärjestelmä. Itse asiassa monissa koneissa käyttöjärjestelmän tärkein tehtävä onkin juuri tarjota säilytystilaa tiedoille, joiden pitää olla tarvittaessa helposti saatavilla.

Ensimmäiset tiedostojärjestelmät tarjosivat käyttäjälleen tuskin muuta kuin tavan varata tietty levyalue, antaa sille numero tai jopa nimi, kirjoittaa alueelle ja lukea sitä. Dynaamisista tarpeen mukaan varattavista tiedostoista ei osattu uneksiakaan.

Seuraavaksi kehitettiin tiedostojärjestelmien ominaisuuksia ohjelmien kannalta. Tuolloin useimmat tietokoneiden käyttäjät kirjoittivat myös ohjelmat itse. Tiedostojen nimeämisen selkeyteen ja tiedostojärjestelmän käyttäjäystävällisyyteen ei vielä kiinnitetty erityistä huomiota. Riitti, että tieto säilyi. Tätä kehitysastetta edustivat 1960-luvulla esimerkiksi XDS-940, CTSS, OS/360 ja TOPS-10 [SPG91, sivut 633–639]

Kun tietokoneiden osituskäyttö yleistyi ja tietokoneita käyttivät yhä enemmän myös ne, jotka eivät olleet tietojenkäsittelyn ammattilaisia, käyttöjärjestelmän tarjoamat suojausominaisuudet tulivat yhä tärkeämmiksi. Samalla kun eri käyttäjien istunnot ja ohjelmien resurssit eristettiin toisistaan, myös pysyvämmälle tiedolle, tiedostoille, piti rakentaa suojaukset muita käyttäjiä vastaan. Jotkin tämän aikakauden ratkaisut olivat nykyisin ajateltuna hyvinkin hankalakäyttöisiä ja kömpelöitä. Usein vanhasta eräajokäyttöjärjestelmästä rakennettiin kyhäelmä, joka erilaisin kikoin pyrki pitämään käyttäjien tiedot erillään.

Kehittyneemmissä käyttöjärjestelmissä ensin niiden tekijät ja myöhemmin käyttäjätkin huomasivat, että siistillä ortogonaalisella rakenteella saadaan aikaan ympäristö, jossa käyttöjärjestelmän eri osat — prosessien hallinta, muistin hallinta ja tiedostojärjestelmä — hahmottuvat koneen käyttäjälle selkeinä. Tällaisessa ympäristössä on helppo etukäteen ymmärtää, mitä kukin aliohjelmakutsu tai komento kussakin tilanteessa tekee. Esimerkkinä omana aikanaan hyvistä käyttöjärjestelmistä voidaan mainita TOPS-20 ja Multics.

3.1 TOPS-20

TOPS-20 on Digitalin 36-bittisen PDP-10 -perheen kehittyneempi käyttöjärjestelmä. TOPS toi ohjelmoijien ulottuville mm. tiedosto-

järjestelmähierarkian sekä tiedostojen kuvaamisen muistiin.

3.2 VMS

Digital hylkäsi PDP-10 -perheen 1980-luvun alussa ja alkoi panostaa VAX-perheeseen. VAX on 32-bittinen arkkitehtuuri, jonka pääasiallisena käyttöjärjestelmänä Digitalin mielestä on VMS. VMS on ohjelmoijan kannalta erittäin monipuolinen käyttöjärjestelmä. Tiedostojärjestelmä tarjoaa ominaisuuksina mm. hierarkian levyjen sisällä, oikeuslistat (ACL) ja tiedostojen kuvaamisen muistiin. VMS:n suuri heikkous on tiedostojen nimeämisen epäjohdonmukaisuus. Tiedostonimissä on käytössä useita erikoismerkkejä ja niiden yhdistelmiä.

VMS:n täydellinen tiedostonimi on muotoa: [VMS88]

solmu: :*laite*: [*hakemisto*] *nimi*. *tyyppi*; *versio*

solmu on sen DECNET-koneen nimi, jossa tiedosto sijaitsee. Jos kyseisen tiedoston käsittely vaatii normaalista poikkeavia oikeuksia, lisätään solmun nimen perään lainausmerkeissä välilyönneillä erotettuna käyttäjätunnus ja sitä vastaava salasana.

laite voi olla joko looginen laitenimi tai fyysinen laitenimi. Loogiset laitenimet ovat joko koko järjestelmälle yhteisiä nimiä tai käyttäjän omia. Loogisia nimi käytetään määrittelemään sellaisia laitteiden tai hakemistojen nimiä, jotka saattavat muuttua.

hakemisto kuvaa laitteen alla olevaa hakemistohierarkiaa, jossa allekkaiset hakemistojen nimet erotetaan pisteillä.

nimi on enintään 39 merkkiä pitkä. Sallittuja merkkejä ovat kirjaimet A–Z, numerot 0–9, alaviiva _, tavuviiva - ja dollarimerkki $.

tyypillä on sama syntaksi kuin nimelläkin.

versio on kokonaisluku 1–32767.

Täydellinen tiedostonimi voi VMS:ssä olla siis vaikkapa:

```
HUBBUB"JONES PANDEMONIUM"::DISK1:[JONES.PAYROLL]STAFF_SALARIES.TXT;666
```

Tiedostonimessä esiintyy runsaasti erikoismerkkejä, ja sen jäsentely ei ole johdonmukaista. Syynä on VMS:n polveutuminen aikaisemmista Digitalin käyttöjärjestelmistä.

3.3 MS-DOS

MS-DOS on Microsoftin Intelin 80x86 -prosessoriperheelle kehittämä henkilökohtaisten tietokoneiden käyttöjärjestelmä. MS-DOSin tiedostojärjestelmä tarjoaa sangen alkeelliset palvelut ohjelmoijalle. Vaikka hakemistohierarkiaa tuetaankin, mm. tiedostonimiavaruus on hyvin rajoitettu. Vanhemmissa MS-DOSin versioissa oli partitioilla 30 megatavun kokorajoitus. MS-DOS ei myöskään tarjoa minkäänlaista tiedostojen suojausta. Ohjelma voi jopa käyttöjärjestelmästä välittämättä kirjoittaa suoraan levylle ja näin sotkea tiedostojärjestelmän.

4 UNIX-käyttöjärjestelmä

UNIX kehitettiin alkujaan Bellin laboratorioissa, New Jerseyssä 1970-luvulla. Bellin laboratorio luopui Multics-projektista ja muutamat mukana olleet päättivät tehdä oman, selkeän ja yksinkertaisen käyttöjärjestelmänsä. UNIX piti saada toimimaan PDP-11-koneessa. Sen piti siis olla paljon Multicsia pienempi.

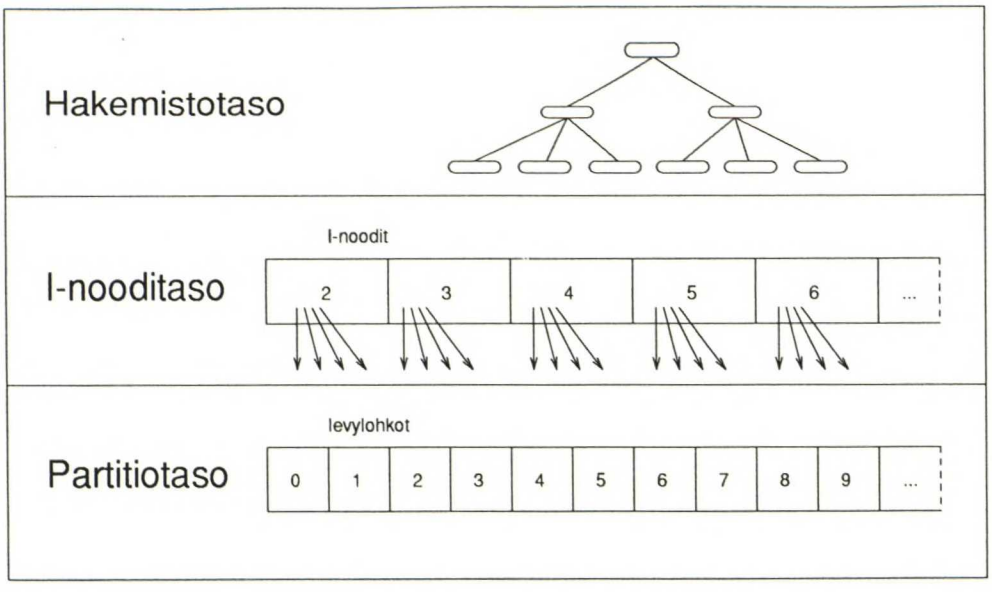
UNIXin tekijät ovat luonnehtineet käyttöjärjestelmäänsä lyhyesti luettelemalla sille tyypilliset piirteet, joita harvoin tapaa (tuon ajan) käyttöjärjestelmissä [RT78]:

- Hierarkkinen tiedostojärjestelmä, joka voi koostua useista eri hakemistopuun paikkoihin kiinnitettävistä partitioista.
- Syöttö ja tulostus, joka toimii samalla tavoin tiedostoille, laitteille ja prosessien väliselle liikenteelle. Tiedosto on aina tavujono.
- Mahdollisuus käynnistää asynkronisia (tausta-) prosesseja.
- Käyttäjän valittavissa oleva komentotulkki.
- Yli sata alisysteemiä sisältäen mm. tusinan verran eri ohjelmointikieliä.
- Helppo kannettavuus.

UNIXissa oli siis toteutettu monia uusia ajatuksia. Koska UNIXin piti olla pieni ja siten ortogonaalinen, tuli siitä myöskin helppo ymmärtää. Tiedostojärjestelmiin liittyvä UNIXin innovaatio oli se, että sekä tiedostot että laitteet näkyivät tiedostojärjestelmässä, joka oli kaiken lisäksi hierarkkinen. Tiedostonnimillä ei ollut muuta kuin pituusrajoitus, sekä muutaman erikoismerkin varaaminen systeemin käyttöön. Lisäksi tiedostoilla oli vain yksi rakenne, tavujono.

4.1 UNIXin tiedostojärjestelmän tasot

Alkuperäisen UNIX 6:n tiedostojörjestelmä oli hieman yksinkertaisempi kuin tässä esiteltävä BSD UNIXin FFS (Fast File System) [McK84]. FFS sisältää kuitenkin ominaisuuksia, jotka ovat oleellinen osa kaikkia nykyisiä UNIX-versioita. Tiedostojärjestelmän tasot näkyvät kuvassa 1.



Kuva 1: UNIXin tiedostojärjestelmän tasot.

Tämä jaottelu kuvaa UNIXin tiedostojärjestelmää staattisesti sellaisena kun se esiintyy levylaitteella. Jatkossa UNIX-semantiikasta puhuttaessa käsitellään erikseen UNIXin toimintaa ajettavien ohjelmien kannalta.

- Tiedostojärjestelmän alimpana tasona on levypartitio, joka on joltakin järjestelmään kuuluvalta levyasemalta tiedostojärjestelmää varten varattu yhtenäinen osa. Tiedostojärjestelmäkoodi näkee partition omana laitteenaan ja sen nimenä on `dev_t` tyyppinen kokonaisluku. Myös käyttäjätason ohjelmat voivat viitata suoraan partitioon `/dev/` -hakemistossa olevan laitetiedoston kautta.

  Tiedostojärjestelmän perusmäärittelyt ovat ns. superblokissa, joka sijaitsee partition alussa. Partitio on jaettu sylinteriryhmiin (cylinder group), joilla on omat superblokkinsa, joissa perusmäärittelyt toistuvat. Toistamisella helpotetaan tiedostojärjestelmän korjaamista häiriötilanteiden jälkeen. Superblokissa on tieto mm. tiedostojärjestelmän koosta, lohkojen koosta, partition lohkojen jakamisesta datalle ja muulle tiedolle.

- I-nooditaso tarjoaa ylös tiedostoabstraktion, jota voidaan käyttää minkä tahansa tiedon säilyttämiseen. Superblokissa ja sylinteriryhmien superblokeissa on tieto kunkin sylinteriryhmän sisältämistä i-noodeista ja siitä, missä partition lohkoissa i-noodien viittaamien tiedostojen tiedot sijaitsevat.

I-noodeihin viitataan niiden numeroilla, joten i-nooditason tarjoama tiedostojärjestelmä on litteä, jossa nimet ovat yhtä suurempia kokonaislukuja.

- Hakemistotaso kuvaa hakemistopuussa olevat symboliset tiedostonimet i-noodeiksi. Tämä toteutetaan käyttämällä i-noodeja hakemistotiedon säilyttämiseen. Kukin hakemistonoodi (i-noodi, jonka tyyppi on hakemisto) sisältää nimi–i-noodinumeropareja, joten kukin hakemisto kuvaa sen sisältämät nimet toisiksi tiedostojärjestelmän i-noodeiksi.

4.2 UNIX-semantiikka

Jatkossa hajautettuja tiedostojärjestelmiä käsiteltäessä viitataan käsitteeseen UNIX-semantiikka. Tällä tarkoitetaan eri UNIXin systeemikutsujen käyttäytymistä ja vaikutusta sekä yhden että useamman prosessin näkemänä. Kun vain yksi prosessi kerrallaan käsittelee tiettyä tiedostoa, on UNIXin systeemikutsujen semantiikka helppo ymmärtää. Mutta kun useampi prosessi käsittelee samaa tiedostoa samaan aikaan (tiedosto on auki useammalle prosessille yhtä aikaa ja ne tekevät tiedostoon liittyviä systeemikutsuja), on tilanne monimutkaisempi.

Vaikka käsittelisimme tavallisia tiedostoja, eli jättäisimme huomiotta laitetiedostot ja putket, on monen prosessin tapauksessa mielenkiintoisia piirteitä. Seuraavien esimerkkien ymmärtäminen edellyttää ohjelmointikokemusta UNIX-ympäristössä.

- Kun prosessit A ja B kumpikin avaavat saman tiedoston F ja vaikkapa A kirjoittaa jotain tietoa johonkin kohtaan tiedostoa, UNIX-semantiikka takaa, että B tämän jälkeen lukiessaan samasta paikasta tiedostoa saa juuri A:n kirjoittaman tiedon. B:n ei siis tarvitse odottaa tiettyä aikaa tai A:n tehdä mitään erityistä, vaan tiedoston sisältö on kullakin hetkellä juuri se, miksi kaikki sitä käsittelevät prosessit ovat sen kirjoittaneet.

- Jos prosessi A avaa tiedoston F ja tekee sen jälkeen `fork()`-systeemikutsun luoden uuden prosessin B, on myös B:lle sama tiedosto auki. Jos A nyt kirjoittaa johonkin kohtaan tiedostoa, siirtyy A:n tiedostokahvaan (file descriptor) liittyvä kirjoitusosoitin kirjoitettua tietomäärää vastaavasti. Jos B nyt kirjoittaa vuorostaan A:lta periytynyttä tiedostokahvaa käyttäen, kirjoittuu tämä tieto A:n kirjoittaman tiedon perään.

UNIXissa periytyneet ja myöskin `dup()`-systeemikutsulla kopioidut tiedostokahvat jakavat mm. kirjoitusosoittimen.

- Jonkin prosessin tekemä tiedoston luonti, poistaminen tai nimen muuttaminen näkyy välittömästi kaikille koneen prosesseille. Yleensäkin kaikki tiedostojärjestelmän operaatiot näkyvät välittömästi.

UNIX on alun perin ollut yhden koneen hajauttamaton järjestelmä. Tässä ympäristössä edellä kuvatun kaltainen toiminnallisuus, jossa eri prosessien kautta tapahtuvan saman tiedoston käsittelyssä on ajallisia riippuvuuksia, ei aiheuta ongelmaa. Hajautetussa ympäristössä, jota käsitellään myöhemmin, UNIX-semantiikan toteutus aiheuttaa pulmia.

4.3 UNIXin ongelmia ja puutteita

4.3.1 Yhtenäinen nimiavaruus

Usein UNIXin innovaationa pidetään ajatusta: "laite on tiedosto". Tämä ei kuitenkaan ole merkittävä asia. Itse asiassa tämä ei edes pidä paikkaansa. Avattu laitetiedosto ei läheskään kaikkien laitteiden kohdalla näytä kovinkaan paljon tavalliselta tiedostolta. Laitteen käyttö vaatii usein erityisten I/O-ohjauskäskyjen antamista jne.

Tärkeätä on sen sijaan yhtenäinen nimeäminen eli se, että kaikki systeemissä olevat oliot nimetään yhtenäisen käytännön mukaisesti. Tällöin mikä tahansa tiedosto tai laite voidaan avata samalla systeemikutsulla, joka palauttaa tiedostokahvan.

4.3.2 Nimiavaruuden merkitys

Käyttäjän kosketus tiedostojärjestelmän nimiavaruuteen syntyy, kun hän huomaa voivansa ryhmitellä asioita eri hakemistoihin ja täten helpottaa niiden löytymistä. Löytymisen helpottuminen johtuu siitä, että komentotulkki on kerrallaan yhdessä hakemistossa, jonka sisältämät tiedostot ovat kerrallaan helposti näkyvissä.

Yleensä tiedostojen ryhmittely hierarkiaksi on luontevaa ja helppoa. Yksi hierarkia ei vain aina riitä tiedostojen ja hakemistojen riippuvuussuhteiden kuvaamiseen.

- Joskus tiedostojärjestelmästä halutaan nopeasti löytää esimerkiksi kaikki jonkun tietyn käyttäjän omistamat tiedostot. Tällöin paras esitystapa olisi käyttää ylimpänä jaottelukriteerinä tiedoston omistajaa.

- Jos jostain ohjelmistosta halutaan tehdä kopio muualle lähetettäväksi, olisi kätevää, jos hakemistohierarkiasta näkyisivät vain lähdekoodia ja muuta ohjelmiston kääntämiseen liittyvää tietoa sisältävät tiedostot.

Yksinkertaisen hierarkian problematiikan toinen ulottuvuus paljastuu, kun laajennetaan tarkastelu tavallisista tiedostoista muihin olioihin, joihin käyttäjät ja ohjelmat haluaisivat päästä käsiksi. Esimerkiksi Bell-laboratorion Plan-9 -käyttöjärjestelmässä lähes kaikki oliot näkyvät tiedostojärjestelmän hakemistohierarkiassa.

- Verkko-yhteyksien ja niihin liittyvien protokollien käynnistys voidaan sijoittaa hakemistohierarkiaan. Jos ohjelma haluaa esimerkiksi avata TCP-yhteyden telnet-porttiin koneessa kampi.hut.fi, se voi avata tiedoston nimellä `/net/ip/kampi.hut.fi/tcp/telnet`. Silloin kun protokollaan liittyvät oletusarvot kelpaavat, tällainen ohjelmalle yksinkertainen tapa on tietenkin hyödyllinen. Jotta mekanismista saataisiin täysi hyöty, pitäisi muutkin muodostettuun yhteyteen liittyvät toiminnot saada kuvattua UNIXin tiedostonimimalliin. UNIXin tapa oletusarvojen muuttamiseen on käyttää `ioctl` -kutsua. Mutta tällöin hyöty yhtenäisestä nimeämisestä paljolti katoaa, koska ohjelmassa täytyy kuitenkin olla tietoa käytettävistä protokollista.

- UNIXin prosessilista voidaan saada näkymään hakemistohierarkiassa sijoittamalla se `/proc` -hakemistoon. Prosessit näkyvät hakemistossa tiedostoina, joiden niminä ovat prosessien numerot desimaalilukuina esitettynä. Ohjelma voi avata haluamansa prosessin ja hallita sitä `read`-, `write`- ja `ioctl`-kutsuilla.

4.3.3 UNIXin piilotettu semantiikka

UNIX piilottaa käyttäjän (ajettavan ohjelman) ulottumattomiin tilatietoa, johon ei pääse suoraan käsiksi. Johonkin tietoon ei pääse ollenkaan käsiksi.

- Prosessilla voi olla kontrollipääte. Tämä tarkoittaa sitä päätelaitetta, johon prosessi mm. pääsee käsiksi avaamalla laitteen `/dev/tty`. Samoin kontrollipäätteeltä tulevat tietyt merkit aiheuttavat signaalin lähettämisen prosessille. Kontrollipääte asettuu automaattisesti siksi päätelaitteeksi joka avataan ensimmäiseksi, jos prosessilla ei ennestään ole kontrollipäätettä. Kontrollipäätteen voi poistaa erityisellä `ioctl`-kutsulla. Kontrollipäätesemantiikka pitäisi periaatteessa ottaa huomioon jokaisessa ohjelmassa, joka avaa tiedostoja. Sinänsä tarpeellinen piirre, istunnot, on toteutettu tavalla, joka on epäjohdonmukainen ja epäselvä.

- Prosessiin liittyy kaksi hakemistoa, työhakemisto ja juurihakemisto. Kun `open` -systeemikutsu tulkitsee polunnimeä, aloitetaan tulkinta juurihakemistosta, jos polunnimi alkaa /:llä, muuten työhakemistosta. Prosessi voi muuttaa työhakemistoaan ja riittävillä oikeuksilla varustettu prosessi myös juurihakemistoaan. Mutta on vaikeaa selvittää, mikä kulloinenkin työhakemisto on.

  Jos prosessi haluaa väliaikaisesti muuttaa työhakemistoaan ja palauttaa sen ennalleen, ainoa tapa on ottaa selville työhakemiston polunnimi `getwd` -kirjastofunktiolla. Tämä voi olla mahdotonta, jos polunnimessä on välissä hakemistoja, joihin prosessilla ei ole riittäviä oikeuksia. Polunnimi voi myös olla niin pitkä, ettei sitä voi suoraan käyttää systeemikutsujen parametrinä. Käytännössä riittäisi, jos työhakemistokahvan voisi väliaikaisesti tallettaa johonkin, jotta vanhan työhakemiston voisi myöhemmin palauttaa.

4.3.4 Monen kirjoittajan ongelma

UNIXissa useampi prosessi voi kirjoittaa samaan tiedostoon samanaikaisesti. Harvat ohjelmat ottavat tätä mahdollisuutta huomioon kirjoittaessaan tiedostoja. Käytännössäkin on täysin mahdollista, että kahden eri prosessin kirjoittamat tiedot voivat mennä tiedostoon lomittain, ja tiedosto voi lopulta sisältää virheellistä tietoa. Tässä tapauksessa sessiosemantiikka (katso 6.4) toimisi paremmin ilman, että ohjelmien tarvitsisi käyttää esimerkiksi lukituksia.

4.3.5 Epäjohdonmukaisuus

UNIX-ohjelma saa tiedostokahvan avatessaan jonkin tiedoston tai hakemiston. Alun perin UNIXissa oli tiedoston tilatietojen kysymiseen vain systeemikutsu `stat`, jolle annetaan tiedoston nimi ja osoite puskuriin, johon tieto talletetaan. Ohjelmissa on hyvin usein tarve selvittää jo avatun tiedoston tilatiedot, ja näin lisättiin uusi systeemikutsu `fstat`, jolle annetaan tiedoston nimen asemesta tiedostokahva. Jos asiaa olisi ajateltu aikaisemmin, ei `stat`ia olisi tarvittu ollenkaan. Vastaavasti myös `chdir` ja `execve` olisi voitu korvata `fchdir`illä ja `fexecve`:illä. Vielä hyödyllisempi olisi systeemikutsu `fopen` (jolla ei ole tekemistä `stdio`-kirjaston kanssa), jolloin työhakemistokäsitettä ei tarvittaisi, koska työhakemisto voitaisiin pitää halutussa tiedostokahvassa. Seuraava taulukko esittää nykyiset kutsut ja ne kutsut, joiden toteuttaminen johdonmukaistaisi UNIXia. Tiedostokahvaa käyttävistä kutsuista vain `fstat` on yleisesti toteutettu.

polun nimeä käyttävä	*tiedostokahvaa käyttävä*
`open(path,flags,mode)`	`fopen(fd,path,flags,mode)`
`stat(path,statbuf)`	`fstat(fd,statbuf)`
`execve(path,argv,envp)`	`fexecve(fd,argv,envp)`
`chdir(path)`	`fchdir(fd)`
`link(name,newname)`	`flink(oldfd,oldname,newfd,newname)`

4.3.6 Tietokanta- ja lokitiedostot

UNIXin perusajatuksia oli toteuttaa kaikki tiedostotyypit samoilla peruskutsuilla. Niinpä tiedostoa avattaessa käyttöjärjestelmälle kerrotaan vain, halutaanko tiedostoon lukea ja/tai kirjoittaa. Järjestelmiä hajautettaessa on huomattu, että käyttöjärjestelmälle olisi hyötyä myös tiedosta, haluaako prosessi havaita tiedostoon tehtävät muutokset, vai riittääkö avaushetken sisältö.

5 Tiedostojärjestelmien ominaisuuksista

Tiedostojärjestelmän, kuten minkä tahansa muunkin käyttöjärjestelmän osan tarkoitus, on tarjota ohjelmille, ohjelmoijille ja lopulta käyttäjille jotain tietokoneen käyttöä helpottavaa. Tämä on hyvä pitää mielessä tarkasteltaessa eri piirteitä. Joskus piirteet tai niiden toteutus on tehty enemmän toteuttamisen kuin käytön helppoutta ajatellen. On helpompi toteuttaa tiedostojärjestelmä, jossa tiedostoja ei voi luomisen jälkeen kasvattaa. Samoin on helpompi toteuttaa litteä kuin hierarkkinen järjestelmä. Ideaalinen tiedostojärjestelmä:

- Säilyttää tiedon ikuisesti. Kun käyttäjä tallettaa jotain tiedostoon, hän olettaa saavansa sen halutessaan nopeasti käyttöönsä.
- Helpottaa tiedon löytämistä. Tiedostojärjestelmän tulisi tarjota tapa liittää talletettaviin tietoihin hakuinformaatiota, kuten tiedoston nimi, tiedon löytämisen helpottamiseksi.
- Helpottaa ohjelmointia. Sovelluksen ohjelmoija odottaa voivansa tiedostojärjestelmän palveluja käyttäen vaivattomasti tallettaa tietoja myöhempää käyttöä varten. Ilman tiedostojärjestelmää ohjelmoija joutuisi mm. hallitsemaan suoraan vaikkapa kiintolevyä tai järjestämään tietojen haun jotenkin.
- Helpottaa varusohjelmien tekemistä. Nykyisillä UNIXin systeemikutsuilla on hyvin vaikeaa tehdä ohjelmaa, joka pystyy ottamaan koneen tiedostojärjestelmistä konsistentin varmuuskopion, josta voitaisiin palauttaa jonkin tietyn hetken tilanne. Tiedostojärjestelmän tulee tarjota tavanomaisen ohjelmointiliitynnän lisäksi kutsut kaiken muunkin tiedostojärjestelmään liittyvän tiedon hallintaan.
- Toimii tehokkaasti. Tällä tarkoitetaan sitä, että tiedostojärjestelmä suoriutuu kohtuullisesti erilaisten pyyntöjen toteuttamisesta. Jotkut ohjelmat hakevat ennalta arvaamattomista paikoista monista eri tiedostoista pieniä tietomääriä. Toiset taas lukevat ja kirjoittavat suuria tiedostoja. Hyvä tiedostojärjestelmä sopeuttaa toimintaansa ja mahdollisia optimointejaan pyyntöjen mukaan.
- Suojaa sekä laitteiston että käyttäjien virheiltä. Laitteistovika ei saisi aiheuttaa tiedon häviämistä eikä kovin pitkää

käytön keskeytymistä. Jos käyttäjä vahingossa poistaa tiedostojaan, tulisi tietojen olla saatavilla kohtuullisen ajan. Jos vahinkoa ei huomata heti, tulisi järjestelmän tukea tietojen palauttamista varmuuskopioilta.

5.1 Tiedostotyypit

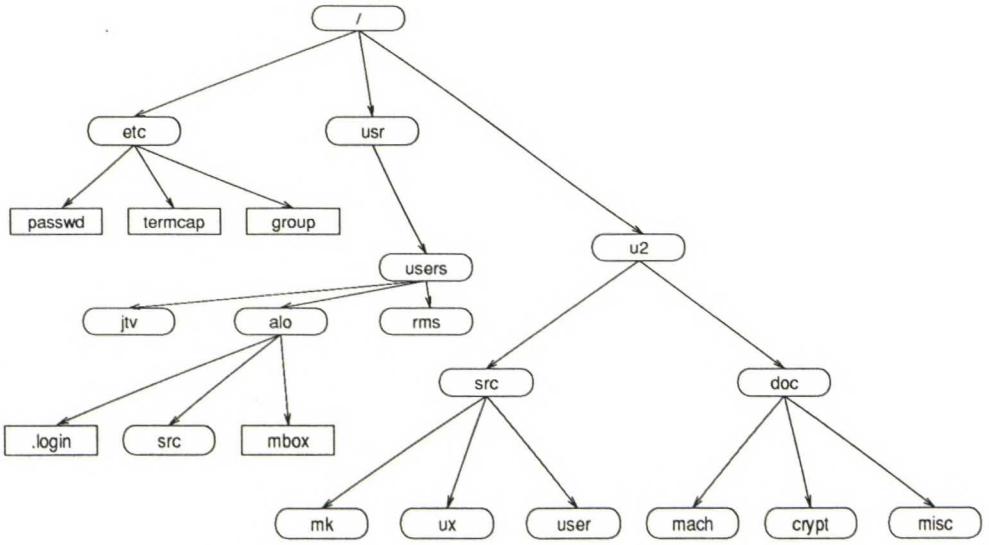
Monet käyttöjärjestelmät, kuten TOPS-20, VMS ja VM/SP tarjoavat useita eri tiedostotyyppejä. Tyypillisiä esimerkkejä ovat kiinteä- ja vaihtuvamittaiset teksti-, binääri- ja tietuetiedostot. TOPS-20 esimerkiksi tukee eri pituisista sanoista koostuvia tiedostoja. Monet käyttöjärjestelmät toteuttavat erilaisia tietokantatyyppisiä hakuja käyttöjärjestelmätasolla. Samoin suoritettava ohjelma on usein oma tiedostotyyppinsä. Ajatus eri tiedostotyppien tarjoamisessa on tiedon tyypin dokumentoinnin lisäksi se, että käyttäjä tai ohjelma ei vahingossa käsittele tietoa väärällä tavalla.

Käyttöjärjestelmätasolla toteutettu tiedostojen tyypitys tekee asiat myös hankaliksi. Mitä enemmän metatietoa käyttöjärjestelmä hallitsee ja sen perusteella rajoittaa tiedostojen käsittelyä, sitä hankalampaa on tiedostoja käsittelevien ohjelmien teko. Ongelma on sama kuin N:n osapuolen keskustellessa erikseen M:n vastapuolen kanssa. Ongelman monimutkaisuus on N:n ja M:n tulo. Mitä vähemmän eri tiedostotyyppejä on, sitä helpompaa on yleiskäyttöisten ohjelmien teko.

Esimerkiksi TOPS-20 -ympäristö hyödynsi tiedostojen tyypitystä monin tavoin. Sen komentotulkki tutki ohjelmaa ajettaessa automaattisesti suoritettavan ohjelmatiedoston, sen linkattavan version ja vastaavan lähdekielisen tiedoston muutosaikoja ja osasi tarvittaessa linkata tai kääntää uuden version ennen ohjelman suoritusta. Vaikka kyseessä onkin komentotulkin piirre, on syytä miettiä, kuinka paljon tietämystä tiedostojen tyypistä millekin käyttöjärjestelmän tasolle halutaan.

5.2 Hakemistorakenne ja tiedostojen nimeäminen

Koska käyttöjärjestelmän alla on tarkoitus pystyä käsittelemään lukuisia eri tiedostoja eri aikoina ja haluttu tiedosto on kyettävä myös löytämään nopeasti, on tiedostojärjestelmän hakemistorakenteella suuri merkitys. Yleisesti hakemistorakenteen tehtävänä on kuvata tiedoston nimi ajossa olevan ohjelman käsiteltävissä olevaan tiedostokahvaan. Tiedostojen haun kannalta on yleensä myös



Kuva 2: Esimerkki UNIXin hakemistopuusta.

tarpeellista pystyä helposti erottelemaan eri käyttäjien tiedostot toisistaan.

Tiedostojärjestelmissä on käytetty monia erilaisia hakemistorakenteita. Yksinkertaisin on yksitasoinen rakenne, jossa on yksi hakemisto, jossa kaikki tiedostot ovat kukin omalla nimellään. Joissakin järjestelmissä on ollut käytössä myös kaksitasoisia rakenteita, joissa päähakemiston alla on tiedostoja sisältäviä alihakemistoja. Kaksitasoinen rakenne on voinut syntyä esimerkiksi vanhan yksitasoisen muuttamisesta useita eri käyttäjiä tukevaksi, jolloin kukin käyttäjä on saanut oman alihakemistonsa omille tiedostoilleen.

Tällä hetkellä yleisin hakemistorakenne on puurakenne, jossa on yksi tai useampia juurihakemistoja, joiden alla on edelleen tiedostoja ja hakemistoja ilman että hierarkian syvyyttä olisi ennalta rajattu. UNIXissa on yksi juurihakemisto, jonka alla voi olla periaatteessa rajoittamattoman syvä hierarkia (katso kuva 2).

Tiedostojen nimeäminen on tiedostojen tyypityksen lisäksi toinen käyttäjälle asti näkyvä käyttöjärjestelmän piirre. Usein tiedostojen nimiin liittyy rajoituksia: Jotkut merkistön merkit ovat kiellettyjä, nimen pituudella on rajoitus, vain isot kirjaimet ovat käytössä jne.

UNIXissa sallittuja merkkejä ovat kaikki 7-bittisen ASCIIn merkit NUL- ja '/' -merkkejä lukuun ottamatta. Pituus on rajoitettu joko 255 merkkiin tai vanhemmissa System V -versioissa 14 merkkiin. Lisäksi kerralla annettavan polun pituudella on rajoitus, joka on yleensä 1023 merkkiä.

6 Hajautetut tiedostojärjestelmät

Edellä käsiteltiin perinteisiä tiedostojärjestelmiä, joissa kaikki operaatiot tapahtuvat yhdessä tietokoneessa yhden käyttöjärjestelmän alaisuudessa. Kuten kappaleessa 4.3.4 havaittiin, jo usean käyttäjän tai prosessin yhtäaikaisesta toiminnasta aiheutuva "hajautus" tuo esiin ongelmia. Yhden koneen ympäristössä ei kuitenkaan ilmene merkittäviä viiveitä tarvittavien lukkojen käsittelyssä, koska kaikki resursseja käyttävät rutiinit pääsevät resursseihin käsiksi yhtä nopeasti.

Hajautetuista järjestelmistä puhuttaessa tarkoitetaan järjestelmiä, jotka on jaettu useisiin tietokoneisiin, jotka ajavat mahdollisesti eri käyttöjärjestelmiä, ovat hyvin eri tehoisia, ja jotka saattavat olla pitkänkin matkan päässä toisistaan. Koneiden väliset tiedonsiirtoviiveet voivat vaihdella muutamasta millisekunnista jopa sekunteihin. Lisäksi koneiden välisessä tietoliikenteessä voi esiintyä pitkiäkin katkoksia, koneet voivat kaatua tai toimia virheellisesti. Kaikki nämä ovat rajoituksia, jotka taas heijastuvat hajautetussa ympäristössä toimivan tiedostojärjestelmän ominaisuuksiin esimerkiksi suorituskyvyn heikkenemisenä, semantiikan muuttumisena tai jopa luotettavuuden huonontumisena.

Jatkossa puhutaan verkkotiedostojärjestelmistä. Käsitteellä tarkoitetaan erityisesti jotakin verkkoa käyttävää tapaa tarjota tiedostojärjestelmäpalveluita verkon yli sekä protokollaa ja mekanismeja, joilla järjestelmä on toteutettu.

Jotta eri verkkotiedostojärjestelmien vertailu olisi mahdollista, on ensin tarkasteltava niihin liittyviä ongelmia ja eri tapoja ratkaista ne. Useimmilla näennäisesti nerokkailla ratkaisulla on varjopuolia, joita esitteet tai myyntimiehet eivät vapaaehtoisesti myönnä.

6.1 Läpinäkyvyys

Useimmissa käyttöjärjestelmissä on tapa siirtää tiedostoja koneesta toiseen joko lähiverkkoa tai muuta tiedonsiirtolinjaa pitkin. Tällöin käyttäjän täytyy tietää missä koneessa tarvittava tiedosto on ja antaa siirtokomento. Verkkotiedostojärjestelmää käytettäessä ei tiedostoa käyttävien ohjelmien käyttäjästä puhumattakaan tarvitse tietää tiedostojen sijaintia, vaan tiedostoihin viittaus tehdään täsmälleen samoin kuin paikallisiinkin tiedostoihin.

Muukaan tiedostojen käytön semantiikka ei saisi muuttua ainakaan siten, että tavanomaisten ohjelmien toiminta muuttuu suuresti. Koneissa halutaan yleensä ajaa valmisohjelmia tai ohjelmiin ei muuten haluta koskea, ja semantiikkaerot eivät saisi aiheuttaa tiedon häviämistä. Joskus läpinäkyvyys katoaa ja käyttäjä saattaa huomata selvästikin tiedostojärjestelmän hajautuksen:

- Yleisin käyttäjän huomaama ero paikallisen ja verkon takana olevan tiedostojärjestelmän toiminnan välillä on tiedon siirrosta aiheutuva viive. Tarvittavien tiedostojen koosta ja käytetyn tiedostojärjestelmän toteutuksesta riippuen ero voi olla suurikin.

- Tavanomaisessa järjestelmässä tiedostojärjestelmä on yleensä kiinteä käyttöjärjestelmän osa. Tiedostojärjestelmän kaatuminen aiheuttaa muunkin käyttöjärjestelmän kaatumisen ja päin vastoin. Hajautetussa ympäristössä palvelimen kaatuminen voi näkyä ohjelmille ja käyttäjälle eri tavoin. Parhaimmassa tapauksessa toiminta jatkuu kuten ennenkin varapalvelimen turvin. Ilman varapalvelinta toiminta voi jatkua palvelimen käynnistyttyä aivan kuin mitään ei olisi tapahtunut. Pahimmassa tapauksessa avoinna olevat tiedostot on avattava uudelleen.

6.2 Tiedostojen jakaminen

Kun kutakin tiedostoa käsittelee vain yksi asiakaskone kerrallaan, kaikki verkkotiedostojärjestelmät selviytyvät hyvin. Mutta tiedostojen jakaminen eri asiakaskoneiden välillä (sama tiedosto on yhtäaikaa auki useammassa asiakaskoneessa) saattaa aiheuttaa odottamattomia seurauksia.

UNIXissa prosessin on mahdollista kirjoittaa avoimeen tiedostoon mihin tahansa kohtaan minkä tahansa määrän tavuja. Koska useimmat verkkotiedostojärjestelmät toteuttavat levylohkojen puskurointia, edellyttää onnistunut tiedostojen jakaminen asianmukaisten lukitusten toteuttamista.

6.3 Puskurointi

Tehokkaan toiminnan saavuttamiseksi tietoa on pakko puskuroida asiakaskoneessa. Hajauttamattomissakin tiedostojärjestelmissä

käytetään puskurointia levyliikenteen vähentämiseksi. Hajautetuissa järjestelmissä puskuroinnin tavoitteena on verkkoliikenteen vähentäminen ja toiminnan nopeuttaminen [SPG91, sivut 491–541].

Puskurointi ei aiheuta tiedostojen jakamisen kannalta ongelmia jos tiedosto on auki vain yhdelle asiakkaalle tai kaikki asiakkaat vain lukevat tiedostoa. Usean asiakkaan kirjoittaessa samaan tiedostoon ongelmaksi muodostuu eri asiakkaiden ja palvelimen tietojen pitäminen keskenään konsistentteina. Ongelmaan on useita ratkaisuja:

- Puskuroidulle tiedolle pidetään yllä lukkoja, joiden avulla varmistetaan, että muutettava tieto poistetaan muualta kuin kirjoittavan asiakkaan puskurista. Asiakas varaa lukon aina ennen tiedosto-operaatiota, tekee operaation ja vapauttaa lukon.

- Toimitaan lukkojen avulla kuten edellisessä vaihtoehdossa, mutta asiakas ei vapauta lukkoa automaattisesti, vaan vasta palvelimen pyytäessä sitä jotakin toista asiakasta varten.

- Kovien tiedostojen käyttö on eräs mahdollisuus. Kova tiedosto on tiedosto, jonka sisältö ei tiedoston perustamisen jälkeen muutu. Kun asiakas luo tiedoston ensimmäisen kerran, se voi kirjoittaa siihen kunnes tiedosto talletetaan palvelimelle. Tällöin tiedostolle annetaan yksikäsitteinen tunniste, jota ei käytetä enää uudelleen. Tiedosto voidaan kuitenkin tallettaa ennen käytetyllä nimellä, mutta talletus ei vaikuta asiakkaisiin, joille tiedoston vanha versio on jo auki. Tätä ratkaisua on käsitelty tarkemmin artikkelissa [SGN85].

Konsistenttiusongelma on itse asiassa kaksiosainen. Jos tiedostoja käytetään tietokantatyyppisesti, eli tiedostoa on tarpeen muuttaa lyhyin väliajoin ja muutosten täytyy näkyä kaikille asiakkaille, joille tiedosto on avoinna, ainoa ratkaisu on lukitusten käyttö. Tällöin lukituspyyntöjen täytyy yleensä tulla sovellusohjelmalta, jotta tiedoston sisällön konsistenssi voidaan säilyttää.

Jos tiedostot ovat luonteeltaan kokonaisuuksia, kuten tekstitiedostoja tai suoritettavia ohjelmia, uusi versio luodaan aina kokonaan uudestaan. Tällöin kovien tiedostojen käyttö on paras ratkaisu, koska ei haluta useamman asiakkaan voivan muuttaa samaa versiota tiedostosta. Jos useampi ohjelma kirjoittaa samanaikaisesti uutta versiota samasta tiedostosta, ovat näin syntyneet versiot

sisäisesti konsistentteja. Tällöin saattaa kuitenkin olla niin, että käyttäjän todella haluama lopullinen versio on jokin yhdistelmä eri versioista, joten on paras jättää nämä versiot näkyville, jotta käyttäjä voi tehdä päätöksen halutusta tiedoston sisällöstä.

6.4 Sessiosemantiikka

AFS:ssä käytössä oleva sessiosemantiikka [SPG91, sivu 376] yksinkertaistaa puskurikonsistenssin toteutusta. Asiakkaan kirjoittaessa tiedostoon tietoa ei yritetäkään saada näkymään muualla välittömästi. Vasta asiakkaan suljettua tiedoston lähetetään muuttunut tieto takaisin palvelimelle. Asiakkaat, jotka tämän jälkeen avaavat saman tiedoston, näkevät muuttuneen tiedon. Ne, joille tiedosto oli jo auki, eivät havaitse mitään muutosta. Jos useampi asiakas kirjoittaa samaan tiedostoon, vain viimeiseksi tiedoston sulkeneen asiakkaan versio jää näkyviin.

Sessiosemantiikka eroaa kovien tiedostojen käytöstä siinä, että vanhaa tiedostoa ei enää voi avata, koska sitä ei ole enää olemassa muualla kuin mahdollisesti joidenkin asiakkaiden puskureissa. Kovia tiedostoja käytettäessä voidaan haluttaessa pitää useita versioita samasta tiedostosta, jolloin vanhatkin versiot voivat olla käytettävissä.

6.5 Tilallisuus ja tilattomuus

Eräs tapa jakaa eri tapoja asiakkaan ja palvelimen vuorovaikutuksen toteuttamiseen on, pitääkö palvelin yllä tilatietoa asiakkaalle auki olevista tiedostoista. Tilallisessa tavassa palvelin säilyttää tietoa siitä, mitä tiedostoja asiakkaalla on auki, ja mitä asiakas on tiedostoille tehnyt. Tilattomassa tavassa asiakas lähettää jokaisessa kutsussa palvelimelle kaiken kutsun toteuttamiseen tarvittavan tiedon.

Tilallisen tavan etuja ovat parempi tehokkuus, lukitusten helppo toteutus ja palvelimen mahdollisuus toimia järkevämmin, tietäähän se tarkemmin, mitä asiakas on tekemässä. Eräs haitoista on suurehko muistin tarve palvelimessa. Palvelimen täytyy pitää muistissaan tietoa kaikista asiakkaille auki olevista tiedostoista. Katkokset verkkoyhteyksissä voivat myös aiheuttaa viiveitä muille asiakkaille, kun palvelin ei saa asiakkaalla olevaa lukkoa nopeasti vapautettua.

Tilattomassa vaihtoehdossa palvelin on periaatteessa yksinkertaisempi. Palvelimella ei ole mitään tietoa siitä, mitä asiakas tiedostolla tekee, joten optimointiin on vähemmän mahdollisuuksia. Palvelimen kaatuminen aiheuttaa vain katkoksen asiakkaiden toimintaan. Palvelimen käynnistymisen jälkeen on yksinkertaista jatkaa siitä mihin jäätiin. Suurin ongelma on se, että tiedostojärjestelmä on luonteeltaan tilallinen. Peräkkäiset kutsut eivät välttämättä ole toisistaan riippumattomia. UNIX-semantiikan toteuttaminen tilatonta palvelinta käyttäen on vaikeaa.

6.6 Tiedostojen toisto (replication)

Jos asiakaskoneille tarjoaa tiedostopalveluja vain yksi palvelinkone, aiheuttaa palvelimen kaatuminen tai verkkokatkos helposti kaikkien asiakaskoneiden pysähtymisen. Tämän välttämiseksi palvelimen tiedostoja halutaan toistaa eli sijoittaa tiedostoista kopioita useisiin eri palvelijoihin. Kun toistettaviksi tiedostoiksi valitaan ne, joita useimmat asiakkaat tarvitsevat, ei varsinaisen palvelimen lyhytaikainen kaatuminen aiheuta niin suurta haittaa kuin ilman toistoa.

Tiedostojen toisto on oikeastaan tiedon puskurointia verkkoon palvelimen ja asiakkaan välille. Siihen liittyvät siis samat ongelmat ja ratkaisut kuin puskurointiinkin.

6.7 Esimerkkejä verkkotiedostojärjestelmistä

Seuraavassa esitellään yleisempiä UNIX-ympäristössä käytettäviä verkkotiedostojärjestelmiä. Yhteisenä piirteenä kaikille voidaan mainita pyrkimys toteuttaa UNIX-semantiikka niin tarkasti, että tavallinen käyttäjä ei koe yllätyksiä vaikka käsiteltävät tiedostot sijaitsisivatkin verkkotiedostojärjestelmän takana.

6.7.1 Sun NFS

NFS (Network File System) [NFS89] on Sun Microsystemsin kehittämä verkkotiedostojärjestelmä. NFS on toteutettu UNIXiin siten, että asiakaskone voi kiinnittää palvelinkoneen halutun hakemistopuun omaan hakemistopuuhunsa. Sijoittamisen jälkeen palvelimen tiedostot näkyvät asiakaskoneelle periaatteessa samoin kuin paikallisetkin tiedostot.

NFS-protokolla on tilaton. Tämä tarkoittaa sitä, että jokaisessa tiedostonlukuoperaatiossa asiakas lähettää palvelijalle kaiken sen tiedon, joka tarvitaan halutun tietolohkon löytämiseen. Tilattomuus tarkoittaa periaatteessa myös sitä, että erillisten operaatioiden järjestys ei vaikuta toimintaan. Koska NFS suunniteltiin tilattomaksi, se toteutettiin UDP/IP-protokollan päälle. UDP[RFC768] (User Datagram Protocol) on kevyt yksittäisiä paketteja välittävä protokolla, joka ei takaa pakettien saapumisjärjestystä eikä edes niiden perilletuloa tai tietoa mahdollisesta paketin hukkumisesta. Lisäksi Sun jätti ensimmäisissä toteutuksissa myös UDP:n tarkistussumman pois, jolloin edes tiedon oikeellisuutta ei taattu.

NFS:n tilattomuus on kuitenkin vain illuusio, onhan palvelinkoneen hakemistopuun rakenne toki tila, jota asiakas voi muuttaa ja josta operaatioiden tulos riippuu. On myös helppo keksiä kaksi operaatiota, joiden järjestyksen muuttuminen muuttaa ratkaisevasti lopputulosta. Jos asiakas ensin poistaa tiedoston ja sen jälkeen avaa saman nimisen tiedoston, aiheuttaa näiden kahden kutsun järjestyksen vaihtuminen sen, ettei tiedosto jääkään olemaan.

Sun on korjaillut NFS:ää moneen kertaan sen julkistamisen jälkeen. Lukitus tarjotaan eri protokollalla. Puskuroinnin toimivuus on varmistettu mm. aikavalvontaan perustuvilla mekanismeilla. Korjauksista huolimatta NFS edelleen mahdollistaa tilanteet, joissa lopputulos on muuta kuin mitä haluttiin.

6.7.2 Berkeley NFS

BSD-UNIXia kehittävä ryhmä Berkeleyssä on liittänyt BSD 4.3-Reno -versioon Sunin NFS-protokollan vapaan toteutuksen. Berkeley NFS toimii Sunin NFS:n kanssa yhteen ja siinä on lisäksi luotettavuutta BSD-koneiden välisessä liikenteessä parantava piirre: Se voi käyttää UDP:n asemesta myös TCP:tä. TCP takaa tietovirran eheyden tai vikatapauksessa ilmoittaa yhteyden katkeamisen. Toisin kuin UDP:tä käytettäessä tietoa ei voi kadota ilman, että se huomattaisiin. Lisäksi sanomien keskinäinen järjestys säilyy. Tällöin vältytään useimmilta niiltä vaaratilanteilta, jotka edellä mainittiin Sunin NFS:n yhteydessä.

TCP:tä käytettäessä NFS:ään on helppoa lisätä esimerkiksi Spriten (katso 6.7.7) tapaiset lukitukset, joilla taataan UNIX-semantiikka.

6.7.3 System V RFS

AT&T:n UNIX System V sisältää oman verkkotiedostojärjestelmänsä, RFS:n. Toisin kuin NFS, RFS on tilallinen protokolla. Jokaisesta asiakkaalle auki olevasta tiedostosta pidetään tieto sekä asiakaskoneessa että palvelimessa.

Tilallisuus on etu siinä mielessä, että kaikki normaalit UNIXin tiedostonkäsittelyoperaatiot toimivat täsmälleen samoin kuin paikallisia tiedostoja käsiteltäessä. RFS tukee myös laitetiedostoja ilman tilattomuudesta aiheutuvia rajoituksia. RFS on kuitenkin arka palvelinkoneen kaatumiselle tai tietoliikenneyhteyden katkeilemiselle. Yhteyden katketessa asiakaspäässä auki olevat tiedostot täytyy avata uudestaan yhteyden palattua. Valitettavasti juuri mitkään ohjelmat eivät osaa tehdä tätä automaattisesti, jolloin suurikin työ saattaa mennä pilalle.

UNIX-semantiikkaan ei kuitenkaan voi ajatella kuuluvan varautumista muihin kuin levyn täyttymisestä aiheutuviin tiedostojärjestelmävirheisiin, joten RFS:ää ei voida pitää kovin käyttökelpoisena.

6.7.4 CMU RFS

Carnegie Mellon Universityn UNIXiin lisäämä RFS on AT&T:n RFS:stä erillinen toteutus, joka on samoin tilallinen. CMU RFS on suorituskyvyltään heikohko, ja sopii lähinnä satunnaiseen käyttöön. RFS:n kehitykseen ei ilmeisesti ole panostettu enää aikoihin, koska CMU itse käyttää nykyään Andrew File System:iä.

6.7.5 AFS

Andrew File System on CMU:ssa kehitetty verkkotiedostojärjestelmä. AFS on saatavilla kaupallisena tuotteena Transarc Inc:stä useimpiin markkinoilla oleviin UNIX-laitteistoihin.

AFS:n puskurointi toimii tiedostotasolla, eli asiakkaan avatessa tiedoston *koko* tiedosto siirretään asiakkaalle. Tällä vältetään palvelimen ja verkon kuormittumista. Uusimmat AFS-toteutukset tosin osaavat puskuroida myös pienemmissä paloissa. Ratkaisu perustuu siihen havaintoon, että useimmiten luettavaksi avattava tiedosto luetaan kokonaan ja kirjoitettavaksi avattava tiedosto kirjoitetaan kokonaan uudestaan. Poikkeuksen muodostavat loki- ja tietokantatiedostot, jotka täytyy AFS-ympäristössä toteuttaa joko paikallisina levytiedostoina tai muilla sopivilla mekanismeilla.

AFS on hyvin skaalautuva, eli AFS-verkkoon voidaan liittää paljon palvelimia ja asiakaskoneita palvelutason kärsimättä. Mahdollisimman paljon työstä tehdään asiakaskoneessa.

Kun asiakas kirjoittaa tiedostoon, muutokset kirjoittuvat ensiksi vain asiakkaan omassa puskurissa olevaan kopioon. Vasta tiedostoa suljettaessa muuttunut tiedosto lähetetään takaisin palvelijalle. Palvelijalla on tieto asiakkaiden puskureissa olevista tiedostoista. Näin tiedostojen jakaminen on mahdollista kohtalaisen hallitusti.

6.7.6 Amoeba

Amoeba on Alankomaissa Vrije Universiteitissä kehitetty hajautettu käyttöjärjestelmä ja siihen tehty tiedostojärjestelmä [Mul85]. Käyttöjärjestelmänä Amoeban tavoitteiksi asetettiin avoimuus siten, että sitä voitaisiin ajaa erilaisilla ja eritehoisilla verkkoon liitetyillä tietokoneilla. Uudet prosessit sijoitetaan vapaasti verkossa oleviin prosessoreihin, jotka on varattu juuri tähän käyttöön. Käyttöjärjestelmä tarjoaa perusvälineet prosessien hallintaan ja niiden väliseen kommunikointiin. Kaikki muu tehdään tavallisina käyttäjäprosesseina ajettavilla palvelimilla ja niitä käyttävillä sovelluksilla.

Amoeban tiedostojärjestelmäarkkitehtuuri rakentuu usealle eri kerrokselle sijoitetuille palveluille.

- Alimpana tasona on fyysinen taso, jossa on erilaisia tiedon säilytykseen tarkoitettuja laitteita. Näitä voivat olla esimerkiksi magneettiset, puolijohde- tai optiset levyt.
- Lohkotaso käyttää fyysistä tasoa ja tarjoaa palveluna virtuaalilohkojen hallinnan. Lohkotaso tarjoaa ominaisuuksiltaan erilaisia lohkoja. Nopeita, mutta koneen kaatumiselle herkkiä lohkoja voidaan tarjota puolijohdelevyltä. Luotettavia lohkoja voidaan tarjota esimerkiksi monistamalla lohkoja eri fyysisille laitteille.
- Litteä tiedostotaso tarjoaa tavan luoda ja hallita erillisiä tiedostoja. Litteä tarkoittaa tässä sitä, että tiedostoihin viitataan tällä tasolla yksikäsitteisillä mutta kiinteän mittaisilla oliotunnisteilla. Amoeban tiedostot voivat jakaa toistensa lohkoja. Ne voivat myös sisältää viittauksia toisiinsa. Ajatuksena on, että tiedostojärjestelmää käyttävälle sovellukselle

annetaan mahdollisimman paljon vapautta hallita tiedoston rakennetta juuri kyseiselle sovellukselle parhaalla tavalla.

- Hakemistotasolla rakennetaan tuttu hierarkkinen rakenne, joka sisältää nimi-oliotunniste -pareja, joiden avulla itse tiedostoihin päästään helposti käsiksi. Hakemistotaso voi viitata myös muihin kuin Amoeban tiedostojärjestelmän tiedostoihin. Amoebaan on toteutettu myös perinteinen UNIXin tiedostojärjestelmätyyppi.

Amoeban tiedostot (oikeammin tiedostojen versiot) ovat kovia. Tämä tekee puskuroinnin helpoksi. Kun kunkin version muutostiedot hallitaan vain yhdessä paikassa, ei puskurien konsistenssikaan tuota ongelmia. Kun muuttunut tiedosto sitoutuu, muuttuneet tiedot lähetetään palvelijalle, joka tekee muutokset pysyviksi yhdellä kertaa.

6.7.7 Sprite

Sprite on hajautettu käyttöjärjestelmä, joka on peräisin Berkeleystä. Tarkastelen tässä Spriten tiedostojärjestelmän toteutusta.

Käyttäjän kannalta Sprite näyttää UNIXin tavoin yhdeltä tiedostohierarkialta. Se koostuu kuitenkin useista alueista (domain), jotka voivat fyysisesti sijaita eri palvelimissa. Sprite toteuttaa hyvin tarkkaan UNIXin tiedostosemantiikan ja on tilallinen protokolla.

Sprite ei kuitenkaan ole AT&T:n RFS:n tavoin arka tiedonsiirtokatkoksille eikä palvelinkoneen kaatumisille. Spriten skaalattavuuskin on kohtuullinen. Tämä on saatu aikaan säilyttämällä kaikki tilatieto myös asiakaskoneessa. Palvelinkoneen kaaduttua se voi uudelleen käynnistyessään pyytää asiakaskoneilta kaiken sen tilatiedon, joka tarvitaan toiminnan jatkamiseen häiriöttä. Asiakaskoneen kaatuessa palvelin voi unohtaa kyseiseen asiakaskoneeseen liittyvät tilatiedot.

Sprite-tiedostojärjestelmä on oikeastaan vain verkkotiedostojärjestelmäosa Sprite-käyttöjärjestelmän muodostamasta kokonaisuudesta. Lisäksi Spriteen on toteutettu lokitiedostojärjestelmä, joka toteuttaa tiedon talletuksen levylle.

Lokitiedostojärjestelmän idea on lyhyesti, että tiedostojen kirjoittamiseen käytettävä aika pyritään optimoimaan. Kirjoitusoperaatiot yritetään lokalisoida mahdollisimman pienelle aluelle levyllä, jolloin levyn hakuajat pysyvät pieninä.

Sprite-verkkotiedostojärjestelmän suunnitteluun ovat vaikuttaneet useat tietokoneteollisuuden ja -tutkimuksen suuntaukset [OCD+88]:

- Lähes kaikki nykyään hankittavat työasemaluokan tietokoneet liitetään lähiverkkoon, yleensä Ethernettiin. Kaikissa työasemissa on verkkoliitäntä tai se on hyvin halpa.
- Keskusmuistin määrä nykyaikaisissa työasemissa on suuri, yleensä vähintään 8 tai 16 megatavua. Tehokkaammissa työasemissa on jo nyt yleisesti satojen megatavujen keskusmuistit.
- Moniprosessorityöasemat ovat jo nyt käytössä. Tiedostojärjestelmän tulee siis olla rakenteeltaan sellainen, että se toimii jouheasti myös moniprosessoriympäristössä. Tiedostojärjestelmää kuormitettaessa on hyvä, jos se pystyy hyödyntämään koneen prosessoreita rinnakkaisesti.

Spriten mielenkiintoisin piirre on, että se tarjoaa täydellisen UNIX-semantiikan pystyen kuitenkin yleensä puskuroimaan tehokkaasti. Puskureiden konsistenssi ei ole ongelma, jos tiedosto on auki yhtä aikaa vain yhdelle käyttäjälle eli prosessille tai yleisemmin yhdessä koneessa sijaitseville prosesseille. Sprite-palvelin tietää, kuka pitää puskureissaan mitäkin tiedostoa. Niin kauan kun puskuroidaan vain yhdessä paikassa tai tiedostoa ei muuteta, ei mitään erikoista tarvita. Jos joku asiakkaista haluaa muuttaa tiedostoa, täytyy muille puskuroidun tiedon omistajille ilmoittaa, että puskurointia ei enää käytetä. Koska palvelija tietää, kenellä puskuroitu tieto on, ilmoitus on mahdollista lähettää. Asiakkaan saadessa ilmoituksen puskuroinnin lopettamisesta se joko palauttaa muuttamansa tiedon, tai mikäli tietoa ei ole muutettu, tyhjentää puskurin. Kun tiedosto ei enää ole kenellekään auki kirjoittamista varten, voidaan puskurointia taas jatkaa.

Edellä esitetyt järjestelyt toimivat yleensä erinomaisesti niin kauan kuin mikään osallisista koneista ei kaadu. Jos asiakaskone kaatuu, ei palvelijan tarvitse tehdä muuta kuin havaita asia ja merkitä, ettei kyseiselle asiakkalle ole enää tiedostoja auki. Palvelijan kaatuminen sen sijaan on vaikeampi ongelma. Palvelija sisältää useiden, jopa satojen tai tuhansien, koneiden auki oleviin tiedostoihin liityviä tilatietoja, jotka ovat välttämättömiä toiminnan jatkamiseksi häiriöttä palvelinkoneen uudelleenkäynnistymisen jälkeen. Sprite-palvelin tallettaa kaiken tarpeellisen kuhunkin

asiakkaaseen liittyvän tilatiedon myös kyseiseen asiakkaaseen. Kun asiakas palvelinkoneen käynnistyttyä pyytää palvelua, joka edellyttää hävinneen tilatiedon käyttöä, palvelin pyytää ensin tilatiedon asiakkaalta ja pystyy sen jälkeen jatkamaan normaalisti.

7 Mach

Mach on Carnegie Mellon Universityssä kehitetty käyttöjärjestelmäydin. Mach edustaa rakenteeltaan viime aikoina suosituksi tullutta ns. mikroydin-tekniikkaa. Tällä tarkoitetaan sitä, että käyttöjärjestelmäytimessä toteutetaan ne ja vain ne primitiivit, joita ei sen ulkopuolella voida järkevästi toteuttaa.

7.1 Machin historiaa

Machin kehitys aloitettiin 1984. Tavoitteita oli useita, joista osa oli idealistisia ja osa realistisia:

- Pyrittiin määrittelemään rajapinta, joka tarjoaa oliokeskeisen liitynnän järjestelmän perusosiin. Erilaisia perusosia on suhteellisen vähän.
- Järjestelmään pyrittiin saaman hyvä tuki sekä hajautetulle että moniprosessoinnille.
- Kannettavuuteen eli helppoon siirrettävyyten eri laiteympäristöihin kiinnitettiin erityistä huomiota. Järjestelmän tulee olla helposti siirrettävissä eri prosessori-, muisti- ja I/O-arkkitehtuureihin.
- Järjestelmän tulee olla koko kehitysprosessin ajan yhteensopiva Berkeley UNIXin kanssa. Tällä varmistettiin järjestelmän laaja käyttö ja sitä kautta palautteen runsaus.
- Suorituskyvyn tulee olla vertailukelpoinen kaupallisten UNIX-toteutusten kanssa.

Machia on markkinoitu määrätietoisesti vuodesta 1986 lähtien. Aluksi sitä tarjottiin vain amerikkalaisille yliopistoille, mutta muutaman viime vuoden ajan sen on voinut saada ilmaiseksi kuka tahansa AT&T:n UNIX-lisenssin haltija. Nykyisen Mach-version ydin on täysin vapaata koodia, joka ei tarvitse lisenssejä, mutta käyttökelpoiset UNIX-palvelimet tarvitsevat lisenssin vielä toistaiseksi. Yleisimmin käytössä olevan UNIX-palvelin pohjautuu BSD 4.3:een, joka vaatii UNIX-lähdekoodilisenssin. CMU:ssa on tekeillä BSD:n vapaaseen koodiin perustuva UNIX-palvelin. Ensimmäinen versio palvelimesta on jo saatavilla, mutta se ei toimi vielä kovin luotettavasti.

7.2 Mach filosofiana

Machin kehittäjä Rick Rashid on sanonut, että Mach ei ole käyttöjärjestelmä eikä uskonto eikä edes mikään tietty kasa koodia. Mach on enemmänkin joukko ideoita, joita kokeiltaessa on rakennettu toimiva käyttöjärjestelmä, jota jaetaan Machin nimellä. Tässä on paljon totta, sillä Mach on historiansa aikana kirjoitettu pariin kertaan uudestaan.

Machin filosofiaan kuuluu myös olla tinkimättä ominaisuuksista toteutuksen helppouden kustannuksella. Muiden käyttöjärjestelmien kehitysprojekteista havaitaan, että usein hyvältä eikä edes niin vaikeasti toteutettavalta näyttävät ominaisuudet ovat lopulta jääneet tekemättä. Machissa on toteutettu kaikki ne piirteet, jotka alkuperäisissä suunnitelmissa oli.

7.3 Machin perusosat

Machin voidaan ajatella perustuvan kolmen eri osa-alueen vuorovaikutukselle.

- Prosessien hallinta huolehtii säikeiden skeduloinnista ja kehysten resurssien hallinnasta.
- Viestin välitys huolehtii tiedon siirtämisestä säikeeltä toiselle.
- Muistin hallinta tarjoaa prosesseille palvelut muistiavaruutensa hallintaan.

Tässä tekstissä ei kuvata Machin systeemikutsurajapintaa kuin siltä osin mitä myöhemmin esitettävien ohjelmaesimerkkien ymmärtämiseksi on tarpeen. Tarkka kuvaus Machin ohjelmointirajapinnasta on käsikirjassa [CMU91]. Oppaissa [Loe91b, Loe91c] on hyvin kattavasti käsitelty monisäikeisten ohjelmien ja niihin perustuvien palvelimien kirjoittamisessa tarvittavia aliohjelmakirjastoja. Kirjoituksessa [Loe91a] on kuvattu Machin sisäistä rakennetta ja valittuja ratkaisuja.

7.3.1 Kehykset ja säikeet

Prosessien hallinta muodostaa perustan käyttäjien ohjelmien ajamiseen. Kuten muissakin mikroydinratkaisuissa, Machissakin perinteinen prosessikäsite on jaettu kahteen osaan, kehykseen

(task) ja säikeeseen (thread). Kehys sisältää prosessista sen muistin sekä kaikki muutkin sen hallinnassa olevat resurssit. Säie on kehyksen alla oleva ohjelmaa suorittava olio. Säikeeseen kuuluvat koneessa käytettävän keskusyksikön rekisterit, niiden joukossa ohjelmalaskuri ja pino-osoitin. Niinpä säie voidaankin helpoiten ymmärtää virtuaaliprosessorina. Kunkin kehyksen alla voi olla yhtäaikaa toiminnassa nolla tai useita säikeitä.

Kehykset ja säikeet ovat ytimen tarjoamia perusolioita, joiden hallinta tapahtuu porttien avulla. Kutakin kehystä ja säiettä vastaa portti, jonka vastaanotto-oikeus on ytimellä. Jokainen, jolla on lähetysoikeus porttiin, voi hallita vapaasti kyseistä kehystä tai säiettä.

Jatkossa käytetään myös Machin yhteydessä käsitettä prosessi. Prosessi on tällöin yhteisnimitys kehykselle ja joillekin sen alla toimiville säikeille.

7.3.2 Muistin hallinta

Virtuaalimuistin hallinta tarjoaa prosesseille joustavat mahdollisuudet hallita muistiavaruuttaan alla olevan konearkkitehtuurin määräämissä rajoissa. Itse Mach ei aseta mitään rajoituksia sille, mihin kohtaan osoiteavaruuttaan prosessi voi muistia varata. Prosessi tarvitsee muistia esimerkiksi tallettaakseen omia tuloksiaan tai varatakseen tilaa pinoa varten. Tällöin prosessi pyytää Mach-ytimeltä nollattua muistia `vm_allocate` -kutsulla. Joskus prosessi voi tarvita muistialueen, johon halutaan jo valmiiksi jotain tietoa, kuten jonkin muun prosessin tuottamaa tietoa tai vaikkapa jonkin erikseen ladattavan ohjelman osan. Voidaan myös haluta, että muistialueelle kirjoitettu tieto menee talteen myöhempää tarvetta varten. Näissä tapauksissa muistialue voidaan varata kutsulla `vm_map`. Tällöin Mach-ydin sitoo varattavaan muistialueeseen kutsussa annetun *portin*, jolloin muistialuetta ei nollatakaan, vaan prosessin koskiessa muistiin ydin pyytääkin vastaavan muistisivun portin kautta muistialueesta vastaavalta palvelijalta.

7.3.3 Kommunikaatio

Nykyaikaiseen käyttöjärjestelmään kuuluu myös tehokas viestien välitys prosessilta, tai Machin tapauksessa säikeeltä toiselle. Machin viestinvälitys toimii tehokkaasti sekä lyhyille että paljon tietoa sisältäville viesteille. Viesti koostuu aina suhteellisen lyhyestä

perusosasta, jota käytetään viestin perustietojen, viestin osien tyyppitiedon sekä lyhyissä viesteissä myös itse tiedon välitykseen. Jos halutaan välittää suurempia tietomääriä, tyyppitiedossa kerrotaan perusosassa olevan tiedon tyypin ja määrän lisäksi vain osoitin itse tietoon, joka edelleen sijaitsee osoitteen mukaisessa paikassa lähettävän kehyksen osoiteavaruudessa. Tällöin ydin ei kopioi tietoa välittömästi vastaanottajalle, vaan tekee tiedosta virtuaalisen kopion suojaamalla kyseisen muistiavaruuden alueen kirjoittamiselta. Näin sekä lähettäjälle että vastaanottajalle voidaan antaa mahdollisuus lukea samaa muistin sivua, ilman että kumpikaan voi muuttaa tietoa siten, että toinen sen voisi havaita. Vasta jomman kumman kirjoittaessa virtuaalikopioidulle muistisivulle tekee ydin muistisivusta todellisen kopion.

Portti on Machin kommunikaation perusolio. Portti on viestijono, joka on toteutettu ytimen tietorakenteena. Kuhunkin porttiin voi olla olemassa tasan yksi vastaanotto- eli lukuoikeus ja nolla tai useampia lähetys- eli kirjoitusoikeuksia. Kirjoitusoikeuksia on kahta lajia.

- Tavalliseen kirjoitusoikeuteen liittyy viittauslaskuri, joka kertoo, kuinka monta kirjoitusoikeutta kehyksellä on porttiin. Jos kehys vastaanottaa viestissä uuden kirjoitusoikeuden porttiin, johon sillä jo on oikeuksia, viittauslaskurin arvo kasvaa yhdellä. Vastaavasti kehyksen tehdessä porttiin `port_deallocate`-kutsun laskurin arvo vähenee yhdellä. Kun viittauslaskurin arvo laskee nollaan, ei porttiin ole enää oikeutta. Tästä saadaan haluttaessa ilmoitus, jolloin kehys voi vapauttaa porttiin liittämänsä muut tietorakenteet.

- Kertaoikeus (Send-once right) on oikeus, jonka avulla kehys voi lähettää porttiin täsmälleen yhden viestin. Oikeus häviää kun viesti on lähetetty. Kertaoikeudet näkyvät kehykselle aina kukin omilla nimillään. Vaikka kehyksellä olisikin muita oikeuksia samaan porttiin, se ei voi saada tätä millään selville. Kertaoikeuksia käytetään lähinnä vastausportteina etäkutsuissa.

Oikeus on kehyksen resurssi ja sitä edustaa portin nimi kehyksen nimiavaruudessa. Nimet ovat kokonaislukuja. Eri kehysten nimiavaruudet ovat erillisiä ja ydin tarjoaa suojauksen kehyksien välillä, joten kehykset eivät pääse käsiksi toistensa omistamiin portteihin. Jatkossa puhutaan usein portin lähettämisestä viestissä. Tällä tarkoitetaan yleensä portin kirjoitusoikeuden kopioimista ja lähettämistä.

Mach toteuttaa eräänlaisen prosessien välisen kommunikaation (IPC) ja muistinhallinnan välisen dualismin. Machin IPC pystyy välittämään tehokkasti suuriakin muistialueita käyttäen tässä hyväkseen muistinhallinnan ominaisuuksia. Toisaalta muistinhallinta antaa mahdollisuuden yhdistää muistialueita portteihin, Machin IPC-kanaviin, siten että portin takana oleva erillinen prosessi tarjoaa itse muistipalvelun. Tällöin ydin toimii vain muistin puskurina.

7.3.4 Porttijoukot

Mach tarjoaa säikeelle mahdollisuuden vastaanottaa viestejä yhdestä portista kerrallaan. Kun Mach-ympäristössä tehdään erilaisia viestinvälitystä hyväkseen käyttäviä palvelimia, on tapana sitoa kuhunkin käsiteltävään olioon oma portti. Tällä tavoin voidaan olioiden käsittelyoikeuksien hallinta redusoida porttien kirjoitusoikeuksien hallinnaksi. Mach-ydin edelleen huolehtii siitä, ettei porttioikeuksia ole mahdollista varastaa. Palvelimessa, joka huolehtii suuren oliojoukon hallinnasta, on yhtä suuri määrä portteja, joihin saattaa tulla viestejä keneltä tahansa kirjoitusoikeuksien haltijalta.

Tällaista tarvetta varten Machissa on porttien lisäksi mahdollista luoda porttijoukkoja. Porttijoukkoon voi kehys määritellä halutun joukon portteja, joihin sillä on vastaanotto-oikeus. Tämän jälkeen viestin vastaanotto porttijoukosta palauttaa seuraavan viestin, joka tulee johonkin joukkoon kuuluvaan porttiin. Tyypillisessä palvelinohjelmassa on joukko säikeitä, jotka kaikki odottavat viestin saapumista johonkin palveltavan porttijoukon porteista. Kun viesti saapuu, tarvittava laskenta tai muu toimenpide tehdään, ja vastaus lähetetään takaisin viestissä olleeseen vastausporttiin. Vaikka muut kehyksen säikeistä olisivatkin odottamassa viestiä porttijoukosta, voi toinen säie halutessaan lisätä tai poistaa portteja tästä porttijoukosta.

Viestien vastaanottaminen vaihtoehtoisista porteista on tehokasta porttijoukon avulla verrattuna kaikkien porttien luettelemiseen vastaanottokutsussa. Porttijoukkoon kuuluvat portit ovat ytimen tietorakenteessa, eikä niitä tarvitse erikseen etsiä niiden nimien perustelle jokaisen vastaanottokutsun yhteydessä. Koska porttijoukon avulla vastaanottaminen useammasta portista on mahdollista, ei Machiin ole toteutettukaan mahdollisuutta luetella vastaanottokutsussa usempaa kuin yksi portti.

7.3.5 Eri osa-alueiden suhde toisiinsa

Edellä luetellut Machin osa-alueet ovat tiiviissä kosketuksessa toisiinsa. Ne käyttävät ja tarvitsevat toinen toisiaan.

- Lähes kaikki Mach-ytimen systeemikutsut tehdään Machin viestinvälitystä käyttäen lähettämällä sopiva viesti kehyksen, säikeen tai muun ytimen olion hallintaporttiin. Tämä tekee myös yleisen kehysten ja säikeiden hallinnan helpoksi.
- Viestinvälitys käyttää muistinhallinnan palveluita tehdäkseen isojen viestien välityksen tehokkaasti.
- Viestinvälitys on yhteydessä säikeiden skedulointiin. Yhden prosessorin järjestelmässä on esimerkiksi yleensä järkevää antaa ajovuoro juuri sille säikeelle, jolle viesti on menossa.
- Muistinhallinta käyttää viestinvälitystä tarjotakseen mahdollisuuden liittää muistialueisiin muistipalvelijoita. Tällöin ydin saattaa esimerkiksi pyytää palveluita käyttäjätason prosessilta.

7.4 Miten Mach eroaa tavallisista käyttöjärjestelmistä?

Tavanomaiset käyttöjärjestelmät ovat monoliittisia kokonaisuuksia, joiden eri osat ovat vapaassa riippuvuussuhteessa toisiinsa. Käyttöjärjestelmän kehittyessä tästä seuraa ennen pitkää kokonaisuus, jossa muutosten tekeminen on hyvin vaikeaa ilman toimintavirheitä, tehokkuuden oleellista heikkenemistä tai turvallisuusongelmia käyttöjärjestelmän muissa osissa.

Mach tarjoaa käyttöjärjestelmän tekijälle hyvät työkalut hajautettuun olio-ohjelmointiin [Dra91, sivu 2]. Merkittävää on se, että Machin ominaisuuksien käyttö ei huononna suorituskykyä. Kaikkein yksinkertaisimman asian toteuttaminen saattaa Machin primitiivejä käyttäen olla tehottomampaa kuin perinteisen käyttöjärjestelmän tapauksessa, mutta kun ohjelma on vähänkin suurempi, ei ero ole enää merkittävä.

Machin portit täyttävät olio-ohjelmoinnin vaatimukset oliolle sangen hyvin. Portti on hyvin puhdas tietoabstraktio, koska sille ei voida tehdä muuta kuin lähettää siihen viesti tai lähettää se viestissä johonkin muuhun porttiin. Portista ei voi saada selville, mihin

siihen lähetetyt viestit menevät. Portteihin ei liity mitään piilotettua semantiikkaa kuten esimerkiksi UNIXin tiedostokahvoihin (file descriptor) (kappale 4.3.3).

Myös Machin prosessioliot eli kehykset ja säikeet ovat olioita. Niitäkin hallitaan porttien välityksellä. Kuhunkin kehykseen ja säikeeseen liittyy portti, johon lähettämällä kehyksen tai säikeen tilaa voidaan hallita.

Roskan keruu on helppo järjestää olioille, jotka on kuvattu porteiksi. Mach-ytimeltä voidaan pyytää ilmoitus, jos portti joko kuolee (eli siihen ei ole enää lukuoikeutta) tai siihen ei ole enää yhtään viittausta (eli kirjoitusoikeutta). Tällöin porttia vastaava olio ja siihen liittyvät resurssit voidaan vapauttaa tai tehdä jotain muuta sopivaa.

Machin portit ovat luotettavia. Tämä tarkoittaa sitä, että porttiin voi saada kirjoitusoikeuden vain siten, että ydin sen jollekin kehykselle antaa. Ydin taas välittää kirjoitusoikeuden vain, jos joku on sen lähettänyt tai kehyksellä itsellään on oikeudet tähän toimenpiteeseen. Oikeuden saaminen esimerkiksi arvaamalla jokin tunnisteen ei onnistu. Tässä suhteessä Mach eroaa kykypohjaisista käyttöjärjestelmistä kuten esimerkiksi Amoeabasta.

Mach tarjoaa hyvän pohjan rakentaa luotettava ja modulaarinen oliopohjainen käyttöjärjestelmä.

7.5 MIG

MIG on Mach-ympäristössä käytetty työkaluohjelma, jolla tietotyyppi- ja aliohjelmakuvauksista voidaan automaattisesti tehdä tarvittavat C-kieliset aliohjelmapalat sekä palvelin- että asiakaspuolelle. Etäkutsuissa käytetyt tietorakenteet ja funktiot kuvataan MIGin kuvauskielellä. Kielessä on primitiivit, joilla kaikkia Machin IPC:n tukemia tietotyyppejä ja niistä edelleen rakennettuja taulukoita voidaan välittää funktion parametreina kumpaankin suuntaan [Loe91b].

MIG muodostaa määrittelytiedostosta kolme tulostiedostoa:

- Käyttäjän liityntämoduuli sisältää C-funktiot, joiden kutsusyntaksi on kuvaustiedoston määrittelyjen mukainen. Funktiot muuntavat kutsujan antamat parametrit Machin IPC:lle sopivaan muotoon ja edelleen käyttävät Machin IPC-kutsuja. Tästä tiedostosta käännetty tiedosto linkataan asiakas- eli sovellusohjelmaan.

- Käyttäjän otsikkotiedosto sisältää edellisen tiedoston sisältämien funktioiden ja tietotyyppien esittelyt. Tätä tiedostoa käytetään asiakasohjelman kääntämisessä.

- Palvelimen liityntämoduuli käännetään ja linkataan palvelinohjelmaan. Sen sisältämät funktiot erottavat parametrit Machin viestistä ja kutsuvat oikeaa palvelimen funktiota ja lopuksi muuntavat paluuparametrit vastausviestiin sopiviksi ja lähettävät vastausviestin.

Liityntämoduulien avulla asiakasohjelman ei tarvitse tietää mitään Machin viestinvälityksen yksityiskohdista. Etäkutsut näyttävät tavallisilta C-kielisiltä funktioilta. Jos palvelimen halutaan reagoivan erilaisiin erikoistapauksiin, joudutaan näitä varten tekemään jonkin verran koodia. Tähän tarvittava koodi on kuitenkin aina samankaltainen. Seuraavassa luvussa on esimerkkiohjelma, jossa asia on havainnollisesti esitetty.

8 Yksinkertainen asiakas ja palvelin Machissa

Väitin aikaisemmin Machin tukevan oliopohjaista ohjelmointia. Tällöin Machin portteja käytetään kahvoina olioihin siten, että ainoa tapa käyttää olioita on IPC-kutsujen tekeminen vastaaviin portteihin. Esittelen seuraavassa yksinkertaisen olioabstraktion, jossa oliot ovat kokonaislukarvon sisältäviä muuttujia, joiden arvoja voi asettaa ja lukea. Kunkin niminen muuttuja on olemassa niin kauan kuin siihen on olemassa viittauksia jostakin, eli jollain palvelimen ulkopuolisella ohjelmalla on lähetysoikeus vastaavaan porttiin.

Kuten myöhemmin nähdään, tiedostojärjestelmäliitynnän kuvaaminen Machin portteja käyttäen pohjautuu samoihin periaatteisiin. Tiedostojärjestelmäpalvelin on siis rakenteeltaan samankaltainen tässä esiteltävän palvelimen kanssa, vaikkakin luonnollisesti monimutkaisempi. Tämän luvun tarkoituksena on helpottaa Machin ja MIGin käyttöä toimivan esimerkin avulla.

8.1 Määrittelytiedostot

Ohjelmat kääntävä Makefile on seuraava:

```
all : tserver tclient

CFLAGS= -g

EXTRA=  cksys.o quit.o strsave.o
SEXTRA= sym.o port_sleep.o empty_message.o

tserver : tserver.o tserverServer.o objectServer.o $(SEXTRA) $(EXTRA)
        $(CC) -o $@ tserver.o tserverServer.o objectServer.o $(SEXTRA) \
        $(EXTRA) -lthreads -lnetname -lmach

tclient : tclient.o tserverUser.o objectUser.o $(EXTRA)
        $(CC) -o $@ tclient.o tserverUser.o objectUser.o \
        $(EXTRA) -lnetname -lmach

tclient.o : tserver.h object.h

tserver.h tserverUser.c tserverServer.c : tserver.defs
        mig tserver.defs

object.h objectUser.c objectServer.c : object.defs
        mig object.defs
```

MIG-kutsu, joilla asiakasohjelma saa kahvan eli portin tietyn nimiseen muuttujaan, esitellään tiedostossa `tserver.defs`:

```
subsystem tserver 2783600;

serverprefix do_;

#include <mach/std_types.defs>
#include "tserver_types.defs"

/*
 *      Get an object port.
 */

routine tserver_lookup_byname(
                server          : mach_port_t;
                name            : o_name_t;
        out     object          : mach_port_t =
                                        MACH_MSG_TYPE_MAKE_SEND);
```

Kutsussa käytetty tietotyyppi eli merkkijono, jolla välitetään muuttujan nimi, esitellään tiedostossa `tserver_types.defs`:

```
type o_name_t           = (MACH_MSG_TYPE_STRING_C, 8*128);
import "tserver_types.h";
```

Vaikka tyypit tässä yksinkertaisessa esimerkissä voitaisiinkin aivan hyvin esitellä samassa tiedostossa kutsujen kanssa, on selvempää, että tietotyypit eristetään omiin tiedostoihinsa.

Muuttujan arvon muuttamis- ja lukukutsut esitellään tiedostossa `object.defs`:

```
subsystem object 2783700;

serverprefix do_;

#include <mach/std_types.defs>

/*
 *      Operations on an object port.
 */

routine object_change(
                object          : mach_port_t;
                value           : int);

routine object_query(
                object          : mach_port_t;
        out     value           : int);
```

Näiden määrittelyjen lisäksi tarvitaan ainoastaan palvelin- ja asiakasohjelmat sekä mahdollisia apurutiineja sisältäviä tiedostoja.

8.2 Palvelinohjelma

Palvelinohjelma `tserver` tarjoaa periaatteessa rajoittamattoman määrän kokonaislukumuuttujia, joiden arvoja voidaan lukea ja muuttaa. Eri muuttujiin kohdistuvat operaatiot ovat muuttujien luomisen jälkeen toisistaan riippumattomia, joten ne voidaan rinnakkaistaa, kunhan kuhunkin muuttujaan liitetään lukko, jolla varmistetaan, että kutakin muuttujaa käsittelee vain yksi säie kerrallaan. Palvelimelle annetaan käynnistettäessä komento-optiona muuttujia palvelemaan käynnistettävien säikeiden maksimimäärä. Palvelin käynnistää kuitenkin aluksi vain yhden säikeen. Säikeiden kulloisestakin kokonaismäärästä (`nservers`) ja operaatiota suorittavien säikeiden määristä (`nservers_busy`) pidetään kirjaa ja uusi olioportteja lukeva säie käynnistetään, jos määrät ovat yhtäsuuret ja jos ajossa olevien säikeiden määrä on pienempi kuin säikeiden maksimimäärä.

Ohjelmassa on myös tuki sille, että säikeitä poistetaan, jos niitä ei enää tarvita. Nykyinen `cthreads`-kirjasto ei kuitenkaan poista vastaavia Machin säikeitä, joten tästä ominaisuudesta saatava hyöty rajoittuu mekanismin esittelyyn.

Tyypillinen Machissa ajettava palvelinohjelma siis:

- Luo portin, josta palveluja voidaan pyytää. Esimerkin tapauksessa portista voidaan pyytää portteja muuttujiin.
- Rekisteröi portin Machin nimipalvelijalle (netname server), jotta muut voivat käyttää palvelua. `Tserver` rekisteröi itsensä nimellä `tserver`. Nimipalvelija ei nimessä tee eroa (englannin) pienten ja isojen kirjaimien välillä.
- Luo porttijoukon, johon se lisää kaikki portit, jotka viittavat hallittaviin olioihin. `Tserver` lisää porttijoukkoon kaikkien uusien muuttujien portit.
- Tekee kaikki tarvittavat tietorakenteiden alustukset.
- Käynnistää vähintään yhden säikeen jokaista porttia tai porttijoukkoa varten, johon tulevia kutsuja halutaan palvella.

MIG luo jokaisesta kutsuja sisältävästä määrittelytiedostosta funktion, joka purkaa porttiin tulleen viestin ja kutsuu oikeata palvelimen funktiota. Esimerkissä nimipalvelijaan talletettuun porttiin ei odoteta muita kuin MIG-tiedostossa esiteltyjä kutsuja. Tällaisessa tapauksessa riittää, että porttia palveleva säie kutsuu funktiota

`mach_msg_server` antaen parametrinä MIGin luoman funktion nimen, suurimman sallitun viestin koon ja portin nimen.

Olioihin viittaavia portteja palvelevat säikeet joutuvat käyttämään apufunktiota, joka tutkii saapuneen viestin tarkemmin kuin MIGin luoma funktio. Apufunktio, `tserverissä` `object_demux` tekee useita asioita:

- Ensin tarkistetaan, tarvitaanko lisää säikeitä ja onko säikeiden maksimimäärä saavutettu. Jos uusi säie tarvitaan, se käynnistetään.
- Porttia vastaavaan olioon luodaan uusi viittaus eli sen viitauslaskuria kasvatetaan yhdellä. Tällä estetään olion katoaminen kesken suoritettavan operaation.
- Odotetaan, että muiden säikeiden samaan olioon tekeillä olevat operaatiot valmistuvat.
- Tarkistetaan, oliko viesti etäkutsu vai ilmoitus siitä, ettei olioon enää viitata palvelimen ulkopuolelta. Ilmoitus tunnistetaan siitä, että se saapuu kertaoikeuteen lähetettynä toisin kuin etäkutsu, joka saapuu tavalliseen lähetysoikeuteen lähetettynä.
- Tehdään kutsua vastaava operaatio tai tarvittaessa vapautetaan olio.
- Lopuksi vapautetaan alussa varattu viittaus.

`tserver.c`:n listaus on liitteessä B.

8.3 Asiakasohjelma

Palvelujen käyttö asiakasohjelmassa on hyvin yksinkertaista. Asiakas hankkii jostain, vaikkapa nimipalvelijasta portin palveluun. Sen jälkeen porttia käytetään funktiokutsuissa kuten mitä tahansa parametria ilman, että Machin IPC:stä täytyy tietää sen enempää. Jos jotain portin viittaamaa oliota ei enää tarvita, voi viittauksen poistaa `mach_port_deallocate`-kutsulla. Esimerkkiohjelma `tclient` toimii seuraavasti:

- Nimipalvelijalta pyydetään nimeä `tserver` vastaava portti.

- Silmukassa luetaan päätteltä komentoja. `l` *nimi* luo uuden muuttujan ja myöhemmät operaatiot kohdistuvat siihen. `d` vapauttaa yhden viittauksen luotuun muuttujaan. `c` *arvo* asettaa muuttujalle uuden arvon. `q` pyytää muuttujan sen hetkisen arvon ja tulostaa sen.

```
/* This program is based on an example program presented by
   Richard P. Draves at Usenix Mach Symposium at Monterey, California
*/

#include <stdio.h>
#include <mach.h>
#include <mach/message.h>
#include <mach_error.h>
#include <servers/netname.h>
#include "tserver.h"
#include "object.h"

main(argc, argv)
int argc;
char *argv;
{
  mach_port_t server;
  mach_port_t object;
  mach_port_urefs_t object_refs;
  kern_return_t kr;

  /* lookup the port for the t service */

  if (cksys(netname_look_up(name_server_port, "", "TServer", &server),
            "tclient: netname_look_up(TServer)"))
    exit(1);

  /* loop reading and executing commands until eof */

  object = MACH_PORT_NULL;
  object_refs = 0;

  for (;;) {
    char        line[1024];
    int         n;
    char        command[2];
    char        name[1024];
    int         value;

    printf("tclient> ");
    if (!fgets(line,sizeof(line),stdin)) break;
    n = sscanf(line,"%1s %s",command,name);
    if (n < 1) command[0] = 0;
    if (n < 2) name[0] = 0;

    switch (command[0]) {
    case '?':
    case 'h':
      printf("Commands:\n");
      printf("l [name] - lookup object port reference\n");
      printf("d        - deallocate object port reference\n");
      printf("c <num>  - change the value of the object\n");
      printf("q        - query the value of the object\n");
      break;

    case 'l':
      /* lookup a send right for the object port */
```

```
      if (!name[0]) strcpy(name,"default");
      kr = tserver_lookup_byname(server, name, &object);
      if (kr == KERN_SUCCESS) {
        printf("Received a send right for the object port.\n");
        object_refs++;
      } else {
        printf("tclient: tserver_lookup_byname: %s\n",
               mach_error_string(kr));
      }
      break;

    case 'd':
      /* deallocate a send right for the object port */

      kr = mach_port_deallocate(mach_task_self(), object);
      if (kr == KERN_SUCCESS) {
        printf("Deallocated a send right for the object port.\n");
        if (--object_refs == 0)
          object = MACH_PORT_NULL;
      } else {
        printf("tclient: mach_port_deallocate: %s\n",
               mach_error_string(kr));
      }
      break;

    case 'c':
      if (sscanf(name,"%d",&value) < 1) {
        printf("syntax error\n");
        break;
      }
      /* change the value of the object */
      kr = object_change(object, value);
      if (kr == KERN_SUCCESS)
        printf("Changed the object's value to %d.\n", value);
      else
        printf("tclient: object_change: %s\n",
               mach_error_string(kr));
      break;

    case 'q':
      /* query the value of the object */

      kr = object_query(object, &value);
      if (kr == KERN_SUCCESS)
        printf("The object's current value is %d.\n", value);
      else
        printf("tclient: object_query: %s\n",
               mach_error_string(kr));
      break;
    }
  }

  exit(0);
}
```

8.4 Apurutiinit

Machin systeemikutsujen paluuarvojen tarkistuksessa hyödyllinen `cksys`-funktio on tiedostossa `cksys.c`.

```
#include <stdio.h>
```

```
#include <mach.h>

cksys(e,str)

kern_return_t   e;
char            *str;

{
  if (e != KERN_SUCCESS) {
    fprintf(stderr,"%s: (0x%08x:%d) %s\n",str ? str : "error",e,e,mach_error_string(e));
  }
  return e;
}
```

UNIXin sleep-kutsun korvaaja port_sleep sekä kaksi muuta funktiota new_port ja make_wref ovat tiedostossa port_sleep.c.

```
#include <mach.h>
#include <mach/message.h>

/* Allocate a new receive right */

mach_port_t
new_port()

{
  kern_return_t e;
  mach_port_t   port;
  if (KERN_SUCCESS !=
      (e = cksys(mach_port_allocate(mach_task_self(), MACH_PORT_RIGHT_RECEIVE,
                                    &port),
                 "mach_port_allocate"))) {
    return 0;
  }
  return port;
}

/* Make a send right from a receive right */

mach_port_t
make_wref(p)

mach_port_t     p;

{
  kern_return_t e;
  if (KERN_SUCCESS !=
      (e = cksys(mach_port_insert_right(mach_task_self(), p, p,
                                        MACH_MSG_TYPE_MAKE_SEND),
                 "mach_port_insert_right"))) {
  }
  return p;
}

/* Sleep msecs microseconds using receive from an empty port */

port_sleep(msecs)

int     msecs;

{
```

```
  static mach_port_t    sleep_port = 0;
  kern_return_t e;
  struct imsg {
    mach_msg_header_t   hdr;
    mach_msg_type_t             port_desc_1;
    mach_port_t         port_1;
  } imsg;
  if (!sleep_port)
    sleep_port = make_wref(new_port());
  e=mach_msg(&imsg.hdr, MACH_RCV_MSG|MACH_RCV_TIMEOUT,
             0, sizeof imsg.hdr, sleep_port,
             msecs, MACH_PORT_NULL);
}
```

Muuttujien nimitauluabstraktio on toteutettu moduulissa `sym.c`, jonka listaus on liitteessä C.

Esitetty palvelinohjelma on vähäisin muutoksin hyödynnettävissä minkä tahansa palvelun toteuttamisessa. Palvelimeen kannattaa yleensä lisätä testausta helpottavia piirteitä.

- Jos todellisen palvelun operaatiot voivat viedä merkittävästi aikaa, kannattaa palvelimen testiversiossa lisätä operaatioihin joko satunnaisia tai parametreista riippuvia viiveitä. `Tserver` odottaa halutun määrän sekunteja, jos muuttujalle sijoitetaan arvo väliltä 9900–9999. Tällä piirteellä voi testata monisäikeisyyden toimintaa.

Mach-ympäristössä toimivassa monisäikeisessä ohjelmassa pitää ottaa huomioon muutamia seikkoja, jotta ohjelmat toimisivat luotettavasti:

- Ohjelman tietorakenteiden lukitus pitää suunnitella huolella, mieluiten etukäteen. Ohjelman lukittumisen esto on tärkeintä, ja tämä saavutetaan päättämällä lukitushierarkiasta, jota sitten noudatetaan.

  Jos ohjelma halutaan rinnakkaistuvaksi, tarvitaan enemmän kuin yksi globaali lukko. Liian monien eri lukkojen käyttämisestä on se haitta, että lukitukseen kuluu liian paljon aikaa suhteessa hyödylliseen työhön. Normaalisti kannattaa menetellä kuten esimerkkipalvelimessa on tehty: harvemmin tehtäville operaatioille käytetään yhtä globaalia lukkoa ja toisistaan riippumattomille tietorakenteille käytetään omia lukkojaan.

  `tserver.c`:ssä on käytetty kaksitasoista lukitusta. Palvelimen globaalia tilaa muutettaessa varataan aina ensin lukko `global_lock`. Tämän jälkeen voidaan varata tarvittaessa

jonkin olion lukko `object->o_lock`. Jos globaalia tilaa ei muuteta tai sen konsistenssi ei ole tarpeen operaation tekemiseksi, voidaan varata vain käsiteltävän olion lukko. Mutta jos olion lukko on jo varattu, ei globaalia lukkoa voida varata, koska tästä voisi seurata lukittuminen. Lukitusta tarvitsevaan globaaliin tilaan kuuluvat olioiden lista ja niiden nimitaulu. Globaali lukko on varattava aina nimitaulua käytettäessä tai muutettaessa. Olion lukko on varattava aina olion sisältämää tietoa muutettaessa. Lukituskutsut voi löytää `tserver.c`:n listauksesta etsimällä kutsut funktioihin `mutex_lock` ja `mutex_unlock`.

- Monet UNIXin kutsuista ja kirjastofunktioista toimivat väärin tai puutteellisesti. Syöttö- ja tulostusfunktioiden kutsut kannattaa suojata lukoilla, koska funktioita ei ole suunniteltu toimimaan monisäikeisessä ympäristössä. Esimerkiksi `sleep`-funktio ei toimi oikein, jos sitä kutsuu samanaikaisesti useampi säie. `tserver`issä sleep on korvattu omalla `port_sleep`-funktiolla.

Asiakasohjelma on vain testiohjelma, jolla voi kokeilla tietyn palvelun toimivuutta. Yleensä uudelle palvelimelle kannattaa tehdä myös asiakasohjelma testausta varten.

9 Machin ominaisuuksia hyödyntävä tiedostojärjestelmä

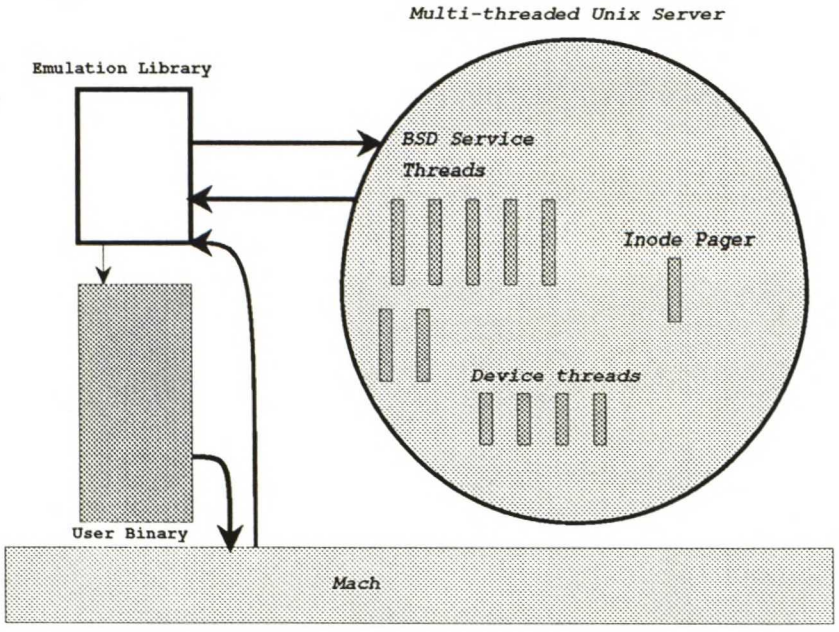
Mach-ydin tarjoaa vain pienen osan niistä palveluista, joita käyttöjärjestelmältä odotetaan. Tämän vuoksi Machin päällä ajetaankin aina versiota jostain tutusta käyttöjärjestelmästä (kuva 3). Saman Mach-ytimen päällä voidaan haluttaessa ajaa samanaikaisesti useampaakin eri käyttöjärjestelmää. Eräs Mach-ytimestä puuttuva osa on tiedostojärjestelmä. Tämä onkin sitten toteutettu käyttöjärjestelmän tarjoamana palveluna. Tilanne, jossa Mach-ytimen päällä ajetaan useaa eri käyttöjärjestelmää, jotka kukin tarjoavat oman tiedostojäjärjestelmänsä, herättää kysymyksiä:

- Tietoa halutaan yleensä jakaa eri käyttöjärjestelmien välillä. Parasta olisi, jos käyttöjärjestelmät näkisivät saman tiedostojärjestelmäpuun, vaikkakin kukin omalla tavallaan.
- Mach tarjoaa mekanismit tehokkaaseen tiedon siirtoon eri kehysten välillä. Ominaisuuksia kannattaisi ehkä käyttää käyttöjärjestelmien omien mekanismien asemesta.

Mach-ympäristö tarjoaa MIGin, jolla tietotyyppejä ja etäkutsuja voidaan kuvata Machin viestinvälityskutsuiksi. Jos MIGiä käyttäen kuvataan yksinkertainen mutta kuitenkin riittävän ilmaisuvoimainen tapa käsitellä tiedostojärjestelmää, voisivat kaikki Machin päällä ajettavat käyttöjärjestelmät hyödyntää sitä. Jo tällä hetkellä käyttöjärjestelmät käsittelevät Mach-koneen laitteita MIG-kutsujen avulla koskematta suoraan itse laitteisiin. Tiedostojärjestelmälle voitaisiin tehdä sama kuitenkin sillä erotuksella, että tiedostojärjestelmäpalvelin ei olisi Mach-ytimessä vaan sen ulkopuolella. Eri käyttöjärjestelmät voisivat käyttää samaa palvelinta, jolloin sama tiedostohierarkia näkyisi kaikkialle.

9.1 Tiedostojärjestelmän kuvaaminen MIG-kutsuilla

Kun tiedostojärjestelmän käsittely halutaan kuvata MIG-kutsuiksi, on otettava huomioon monia asioita, jotta lopputulos olisi käyttökelpoinen eri käyttöjärjestelmien käytössä.



Kuva 3: UNIX-palvelin (UX) Mach 3.0-ytimen päällä, kuva lainattu julkaisusta [DA92]

- Useimmat käyttöjärjestelmät haluavat hierarkkisen tiedostojärjestelmän. Tämän toteuttaminen ei ole ongelma. Hierarkian syvyydelle tai tiedostojen nimille ei ole syytä asettaa mitään erityisiä rajoituksia.
- Tiedostoihin ja hakemistoihin pitää pystyä liittämään metatietoa, jota käyttöjärjestelmät voivat käyttää omassa toiminnassaan.
- Kutsujen olisi hyvä olla sellaisia, että niiden käyttäminen on helppoa myös sovellusohjelman tekijälle. Vaikka kutsut olisikin aluksi tarkoitettu vain käyttöjärjestelmien kutsuttaviksi, ei tämän tarvitse olla ehdoton sääntö.
- Kutsujen pitää olla niin suunniteltu ja toteutettu, ettei niillä pysty sotkemaan alla olevan tiedostojärjestelmän tilaa.
- Kutsuilla ei saa pystyä tekemään sellaista, johon kutsujalla ei ole oikeutta.

Joitakin valintoja on helppo tehdä. Selvin lienee se, että kahva avoimeen tiedostoon tai hakemistoon kuvataan Machin portiksi. Portti vastaa näin käytettynä täysin UNIXin tiedostokahvaa. Kuten vain

tiedostokahvan omistava UNIXin prosessi voi käyttää kahvaa, vain se Machin kehys, jolla on oikeus porttiin, voi lähettää viestejä siihen. Jotkut UNIXit, kuten BSD, voivat myös lähettää tiedostokahvoja prosessilta toiselle. Machissakin on mahdollista lähettää porttioikeuksia kehykseltä toiselle. Portti antaa oikeuden tehdä operaatioita niillä oikeuksilla, joilla portti aikaisemmin hankittiin. Tiedostokahvan kuvaaminen portiksi ei siis rajoita mahdollisuuksia; se, kuinka tarkasti esimerkiksi UNIX-semantiikka toteutuu, riippuu täysin portin päässä olevan palvelimen ominaisuuksista. Tiedosto on tavujono, johon tiedoston sisällön lisäksi liittyy käyttöjärjestelmästä riippumatonta tilatietoa, kuten esimerkiksi:

- Tiedostojärjestelmätyyppi ilmoittaa, minkä tyyppinen on alla oleva tiedostojärjestelmä. Tyyppi voi olla esimerkiksi UFS (Unix File System) tai NFS (Network File System).
- Tiedoston pituus ilmoittaa tiedoston pituuden oktetteina.
- Tiedoston omistaja ja ryhmä sekä suojausbitit ilmoittavat, ketkä voivat käsitellä tiedostoa. Omistaja ja ryhmä ovat kokonaislukuja, joiden merkitys on tiedostojärjestelmäkohtainen.
- Aikaleimat ilmoittavat, koska tiedostoa on viimeksi luettu, kirjoitettu ja koska tiedoston tilatietoa on muutettu.
- Oikeuslistan avulla voidaan toteuttaa perussuojausta tarkempi suojaus.

Tiedoston tilatiedon lisäksi avoimeen tiedostoon viittaavaan kahvaan liittyy omaa tilatietoa. Tämä tilatieto vastaa UNIXin tiedostotietueen sisältämää tietoa. (UNIXin tiedostokahvasta on enemmän sivulla 51.)

- Tiedosto-osoitin kertoo, mihin kohtaan tiedostoa seuraava luku- tai kirjoitusoperaatio tapahtuu. Kun tiedostosta sitten luetaan tai kirjoitetaan, siirtyy osoitin vastaavasti.
- Optiot kertovat tiedostoon mahdollisesti liittyvistä erikoispiirteistä. Esimerkiksi lokitiedoston tapauksessa halutaan, että kaikki kirjoitettu tieto menee tiedoston perään.

UNIXin ja muiden käyttöjärjestelmien puutteiden perusteella voidaan asettaa muitakin tavoitteita, jotka primitiivijoukon olisi hyvä saavuttaa:

- Olisi hyvä, ettei tiedostonimien syntaksia rajoitettaisi tarpeettomasti. Kaikkien käytettävissä olevien merkkien tulee olla sallittuja myös tiedostonnimissä. Tiedostonnimien syntaksin pitää muutenkin olla johdonmukainen ja selvä.
- UNIXin ja Posixin tiedostonkäsittelyrajapintojen tulisi olla helposti primitiiveillä toteutettavissa.
- Primitiivijoukon tulee tukea hyvän tietoturvan toteuttamista. Oikeuslistojen toteutus tarvitaan, mutta niiden käyttämättä jättämisenkin tulee olla helppoa.
- Hajautettujen tiedostojärjestelmien tuki tulee ottaa huomioon siten, että mukaan ei oteta pakollisina sellaisia toimintoja, joiden toteuttaminen hajautetussa ympäristössä on ylivoimaista. Esimerkiksi toiminnot, jotka edellyttävät tiedostojärjestelmän olemista levyllä, sen olevan kiinteänkokoisen jne. rohkaisevat ohjelmoijia käyttämään niitä, mistä seuraa ongelmia toteutuksen ollessa erilainen. Vaikka tällaiset toiminnot voitaisiinkin hajautetussa käytössä jättää toteuttamatta, on parempi, että kokonaisuus on mahdollisimman yhtenäinen ja johdonmukainen.
- Käyttäjän tulee voida helposti määritellä omia hakumenetelmiään, joilla ohjelmien käsittelemät tiedostonnimet kohdistetaan halutulla tavalla alla olevaan todelliseen tiedostojärjestelmään. Monet UNIXin tiedostojärjestelmän ominaisuuksista ovat erikoistapauksia sellaisesta yleisemmästä mekanismista, jolla voitaisiin toteuttaa monia hyödyllisiä toimintoja. Katso 4.3.5.

Edellisen vaatimusten lisäksi liitynnästä on hyvä tehdä oikealla tavalla kerrostettu. Tällöin käyttöjärjestelmäpalvelimet ja sovellukset voivat käyttää tiedostojärjestelmää kulloinkin sopivimilla tasolla. Huonosti kerrostettua järjestelmää käytettäessä ohjelmista tulee yleensä tehottomia, kömpelöitä ja vaikeita ymmärtää.

Esitän alijoukon MIG-kutsuista, joilla tiedostojärjestelmä voidaan kuvata.

```
routine dir_lookup (
        start_dir       : file_t;
        pathname        : string_t;
        flags           : int;
        mode            : mode_t;
        out do_retry    : retry_type;
        out retry_name  : string_t;
        out result      : file_t);
```

dir_lookup jäsentää tiedoston polunnimeä yhden tason eteen päin. Funktio palauttaa kahvan seuraavaan tiedostoon tai hakemistoon parametrissa result. dir_lookup siis avaa tiedoston tai hakemiston. Tiedosto sulkeutuu automaattisesti, kun porttiin ei enää ole viittauksia.

```
routine dir_readdir (
        dir             : file_t;
        out data        : data_t;
        offset          : int;
        out nextoffset  : int;
        amount          : int);
```

dir_readdir lukee hakemiston sisältöä ja palauttaa sen sisältämien hakemistojen ja tiedostojen nimiä standardoidussa esitysmuodossa.

```
routine dir_mkdir (
        directory       : file_t;
        name            : string_t;
        mode            : mode_t);

routine dir_rmdir (
        directory       : file_t;
        name            : string_t);

routine dir_unlink (
        directory       : file_t;
        name            : string_t);

routine dir_rename (
        olddirectory    : file_t;
        oldname         : string_t;
        newdirectory    : file_t;
        newname         : string_t);
```

dir_mkdir luo hakemiston halutun hakemiston alle halutulla nimellä. dir_rmdir poistaa hakemiston ja dir_unlink muun tiedoston. dir_rename siirtää hakemiston tai tiedoston eri paikkaan ja/tai eri nimelle saman tiedostojärjestelmän sisällä.

```
routine file_seek (
        seek_file       : file_t;
        offset          : int;
        whence          : int;
        out newp        : int);
```

file_seekillä voidaan avoimen tiedoston kirjoitusosoitinta siirtää haluttuun paikkaan.

```
routine file_truncate (
        trunc_file      : file_t;
        new_size        : int);
```

file_truncatella tiedosto voidaan lyhentää halutun mittaiseksi.

```
routine io_write (
        io_object       : io_t;
        data            : data_t;
        offset          : int;
        out amount      : int);

routine io_read (
        io_object       : io_t;
        out data        : data_t, dealloc;
        offset          : int;
        amount          : int);
```

Näillä kutsuilla avoimeen tiedostoon voidaan kirjoittaa tai sieltä lukea halutusta kohdasta. Jos halutaan kirjoittaa tai lukea kirjoitusosoittimen osoittamaan paikkaan, annetaan offset-parametrin arvoksi -1.

Esitetyt kutsut riittävät UNIXin tiedostojärjestelmän kuvaamiseen MIG-kutsuiksi. UNIXin tarvitsemat muut, esimerkiksi laitetiedostoihin liittyvät palvelut, toteutetaan ylemmällä tasolla.

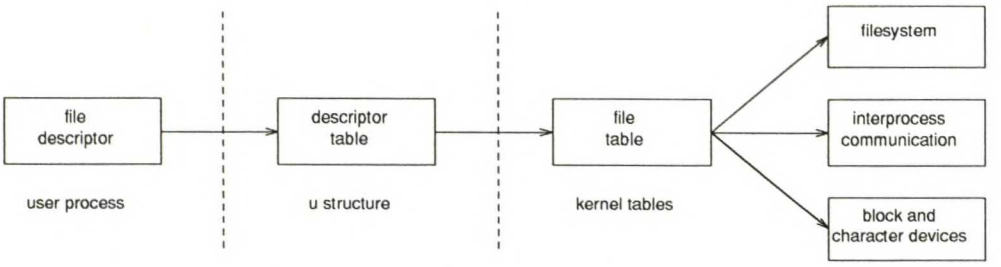
10 MIGiin perustuva tiedostojärjestelmäliityntä UNIXiin

Tavoitteena oli toteuttaa Machin päällä toimivaan UNIX-palvelimeen portteihin perustuva tiedostojärjestelmä. Tällä hetkellä toteutettu tiedostojärjestelmäpalvelin tarjoaa UNIXin tiedostojärjestelmäpalvelut. Tästä on mahdollista kehittää monipuolisempi olemassa olevan toteutuksen kanssa yhteensopiva palvelin. Sivutuotteena syntyi hyödylliseksi osoittautunut testausympäristö, jota on käytetty eri käyttöjärjestelmäpalvelinten testauksessa.

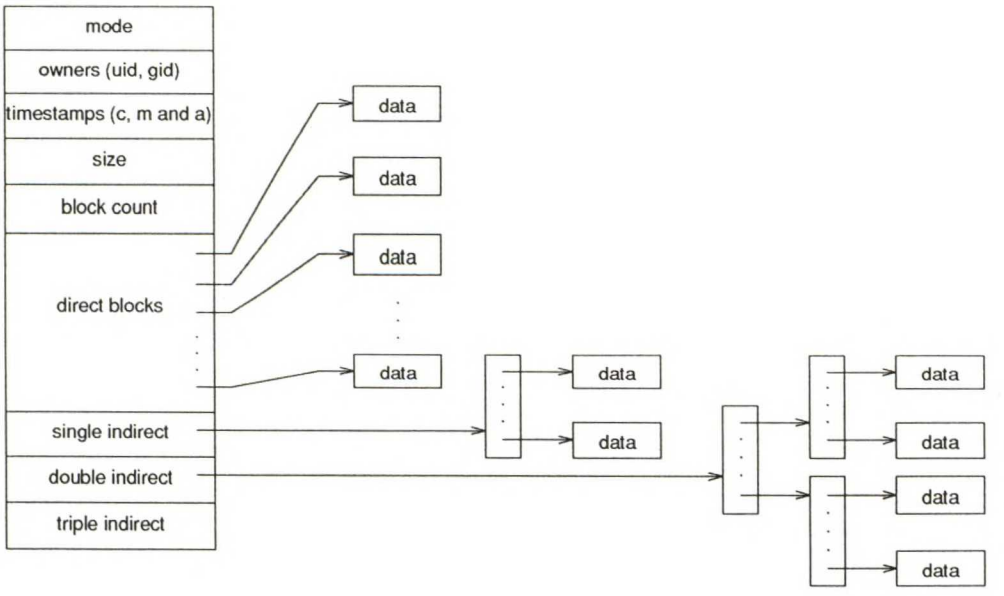
Berkeleyn UNIXeissa ja UNIX System V:ssä Release 3.2:een saakka tiedostojärjestelmä on ollut lähes alkuperäisen UNIX-versio 6:n kaltainen. Avoimeen tiedostoon viittaaminen kulkee usean eri ytimen tietorakenteen kautta, katso kuva 4.

- Prosessin (user process) kahva avoimeen tiedostoon on tiedostokahva, joka on pieni kokonaisluku.
- Tiedostokahva indeksoi ytimen muistiavaruudessa olevassa prosessikohtaisessa u-tietueessa (u structure) olevaa taulukkoa, jossa edelleen on osoitin ytimen tiedostotaulukon (file table) johonkin alkioon eli tiedostotietueeseen (struct file).
- Tiedostotietue sisältää tietoa avoimesta tiedostosta ja sen lisäksi tiedon siitä, minkä tyyppinen tiedosto on. Tällä tiedolla kerrotaan, onko tiedosto varsinaisen tiedostojärjestelmän tiedosto (filesystem) vai esimerkiksi putki, verkkoyhteys (interprocess communication) tai jokin laite (block and character devices).
- Jos kyseessä on tiedostojärjestelmän tiedosto, sisältää tiedostotietue osoittimen tiedoston i-noodiin, jossa onkin kerrottu tiedoston sijainti jonkin levylaitteen partitiossa.

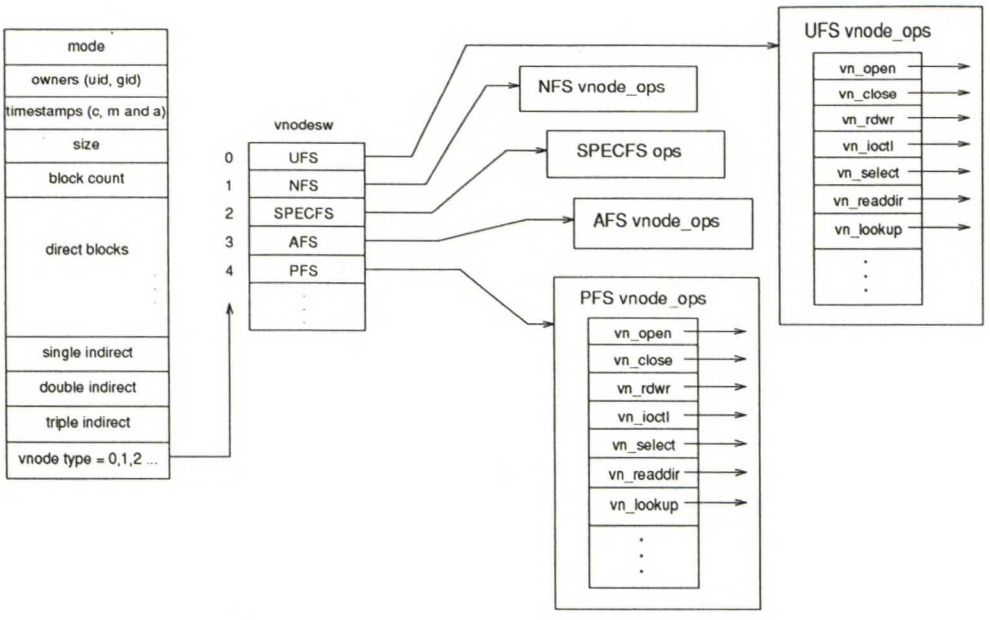
Nykyaikaisten UNIX-toteutusten tiedostojärjestelmärakenne perustuu Sun Microsystemsin kehittämään Virtual File System (VFS) -järjestelmään, jossa tavallisen UNIXin i-nooditaso (katso kuva 5) on korvattu v-nooditasolla, jonka alle taas on sijoitettu joko i-noodirutiinit tai jotkin muut, kuten NFS- tai AFS-rutiinit (katso kuva 6). VFS-taso kehitettiin, koska UNIXin i-noodi pystyi kuvaamaan lohkoista koostuvia tiedostojärjestelmiä, kuten luvussa 4.1 kerrottiin. VFS:ää käyttäen voidaan kuvata periaatteessa



Kuva 4: Avoimen tiedoston viittausketju tiedostokahvasta i-noodiin, kuva lainattu kirjasta [LMKQ89, kuva 6.2]



Kuva 5: Tavanomaisen UNIXin i-noodin rakenne.



Kuva 6: VFS:ään perustuvan UNIXin v-noodin rakenne.

minkälainen tahansa hierarkkiseen hakemistopuuhun perustuva tiedostojärjestelmä.

Uuden VFS-tason alle sijoitettavan tiedostojärjestelmätyypin rutiinien toteuttaminen on huolellisuutta vaativa tehtävä. Käytännössä mikä tahansa ohjelmointivirhe aiheuttaa koko UNIX-palvelimen kaatumisen. Kun tiedostojärjestelmiin liittyviä uusia ideoita halutaan kokeilla, on edellisestä johtuen pakko toteuttaa joku tapa liittää VFS-taso UNIX-palvelimen ulkopuolelle siten, että UNIX-palvelimen ulkopuolisen osan toimintahäiriö ei aiheuta UNIXin kaatumista. Machin portteihin perustuva MIG-liityntä tarjoaa tällaisen tavan. Tähän perustuvan porttitiedostojärjestelmän toteutus on selostettu omassa luvussaan.

Käytössä oleva UNIX-palvelin on perinteinen UNIX siten, ettei se kaikissa mahdollisissa kohdissa voi toimia rinnakkaisesti. Tästä ei-rinnakkaisuudesta johtuen porttitiedostojärjestelmän takana oleva tiedostojärjestelmäpalvelin ei voi luotettavasti toimia saman UNIX-palvelimen prosessina. Näin ollen kokeiluympäristöä varten oli myös luotava mekanismit usean UNIX-palvelimen ajamiseksi saman Mach-ytimen päällä.

10.1 UNIX-palvelin Machin päällä

Käytössä oleva Mach 3.0 on puhdas mikroydin, jossa ei ole mitään perinteiseen käyttöjärjestelmään kuuluvia piirteitä. Jotta järjestelmällä voisi tehdä jotain hyödyllistä, on Machin päällä ajettava jotain varsinaista käyttöjärjestelmää, kuten UNIXia. Käytettävä UNIX-palvelin on monisäikeinen (multithreaded) Machin kehys, jossa monet toiminnot on tehty rinnakkaisiksi. Rinnakkaistamisessa ei ole kuitenkaan menty kovin pitkälle, vaan kaikki vaikeat asiat tehdään edelleenkin vain yhdessä säikeessä kerrallaan.

Vanhojen Mach/UNIX-versioiden alla käännetyt ja linkatut ohjelmat käyttävät sekä Machin että UNIXin palvelujen kutsumiseen systeemikutsuja. Mach 3.0 -ympäristössä kaikki systeemikutsut aiheuttavat kontrollin siirtymiseen ytimelle. Jotta UNIXin systeemikutsut voitaisiin välittää UNIX-palvelimelle, on Machissa piirre, jolla haluttuihin systeemikutsuihin voidaan liittää kutsuneen prosessin osoite, johon ydin edelleen siirtää kontrollin systeemikutsun tapahduttua. Tässä osoitteessa sijaitsee emulointikirjasto (Emulation Library), joka joko suorittaa systeemikutsun edellyttämät toimenpiteet itse, tai edelleen kutsuu UNIX-palvelinta.

10.2 UNIX-palvelin toisen UNIX-palvelimen päällä

Kun UNIX-palvelin käyttää porttitiedostojärjestelmää (PFS, katso luku 10.3) jonkin siihen liittyvän portin kautta, se tapahtuu MIG-kutsulla, joka palaa vasta tiedostojärjestelmäpalvelimen vastattua kutsuun. Jotkin PFS:n kutsuista ovat sellaisia, että koko UNIX-palvelimen tila on lukittava kunnes kutsuun saadaan vastaus. Tästä riippuvuudesta johtuen on systeemin lukittumisen välttämiseksi tiedostopalvelinta ajettava kokonaan PFS:ää käyttävästä UNIX-palvelimesta riippumatta.

UNIX-ympäristön tarjoamat palvelut ovat kuitenkin hyödyllisiä ja testausvaiheessa jopa välttämättömiä tiedostojärjestelmäpalvelimen toiminnalle. Näin syntyi tarve voida ajaa monta UNIX-palvelinta saman Mach-ytimen päällä. Käytännössä tämä järjestetään siten, että UNIX-palvelin käynnistetään toisen UNIXin alta. Katso kuvaa 7.

Jatkossa UNIX-palvelimesta käytetään nimeä UX, ensimmäisestä UNIX-palvelimesta nimeä UX0 ja toisesta nimeä UX1. UX käyttää ympäristönsä kanssa viestimiseen kahta Mach-ytimen tarjoamaa porttia:

- Laiteportti on portti, jonka avulla UX voi pyytää Machilta portteja eri laitteiden käyttämiseksi. Kutakin avointa laitetta kohti on yksikäsitteinen portti, johon tehdyt MIG-kutsut ohjaavat vastaavaa laitetta.

- Koneporttiin tekemillään kutsuilla UX voi hallita Mach-ytimen ja koko koneen tilaa. Kutsuilla voi esimerkiksi asettaa kellonajan ja pysäyttää tai käynnistää uudelleen koko koneen.

Kun UX1:tä ajetaan UX0:n päällä, ei tietenkään haluta, että UX1 voisi vapaasti tehdä kaikkia operaatioita näihin kahteen porttiin. Tämän takia UX1:tä ajetaan erityisen palvelinohjelman (Server server eli SS) alta, joka antaa UX1:lle näiksi kahdeksi portiksi omat porttinsa, jolloin kaikki UX1:n kutsut ohjautuvat SS:lle. SS voi sitten haluttujen määrittelyjen mukaan joko ohjata kutsut sellaisenaan Machille tai tarjota jotkut palvelut emuloiden niitä itse:

- Kun UX1 haluaa avata systeemin konsolin eli laitteen nimeltä `console`, antaa SS portin itse toteuttamaansa palvelimeen, jolla konsolipäätettä voidaan emuloida halutusta paikasta sekä ohjata tulostukset haluttaessa tiedostoon.

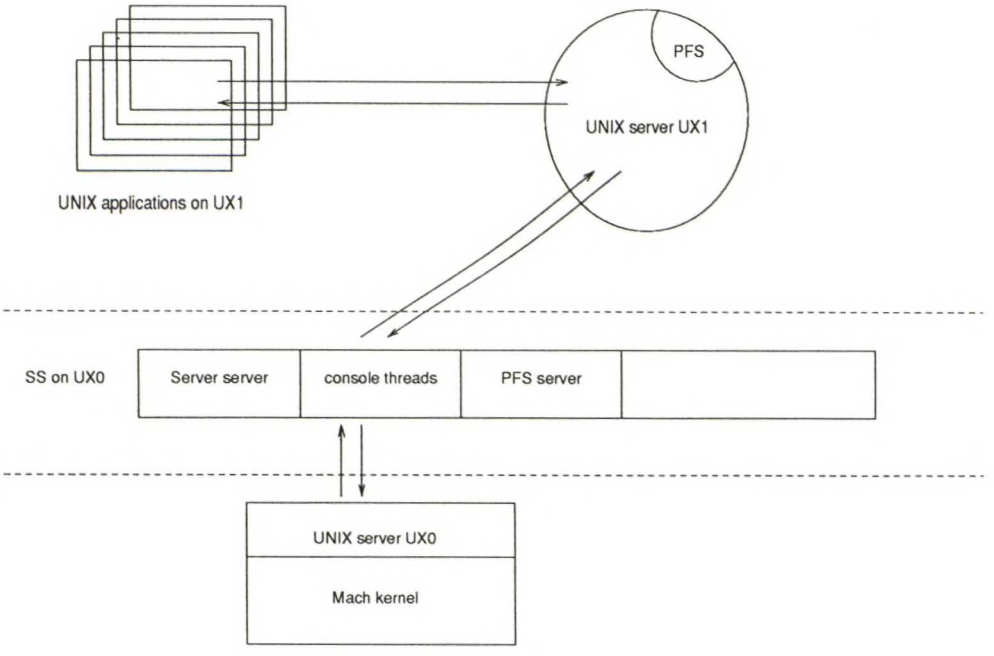
- UX1:n yritykset asettaa kellonaika jätetään huomiotta. Niistä tulostetaan kuitenkin ilmoitus.

- Kun UX1 haluaa pysäyttää tai käynnistää koneen, ei UX1:tä enää haluta ajaa, jolloin vastaava kehys lopetetaan.

- Kun laiteporttiin tulee pyyntö avata laite, jonka nimi on `pfs-*`, SS antaa vastauksena portin, jonka se hakee vastaavalla nimellä nimipalvelijasta. Tätä piirrettä käytetään PFS-porttien välittämiseen UX0:lta UX1:n alla ajettaville ohjelmille.

10.3 Porttitiedostojärjestelmä

Porttitiedostojärjestelmä on tapa kuvata VFS-tason kutsut MIG-kutsuiksi. VFS:n kutsut on lueteltu liittessä A. Kutsut voidaan jakaa kahteen osaan:



Kuva 7: UX1 ajetaan UX0:n päällä

- Tiedostojärjestelmäkutsuilla hallitaan koko tiedostojärjestelmän tilaa, kuten esimerkiksi kiinnitetään tai poistetaan tiedostojärjestelmä, haetaan sen juurinoodi jne.
- Noodikutsuilla tehdään yksittäisiin hakemistoihin tai tiedostoihin liittyvät toiminnot. Näillä luetaan ja kirjoitetaan tiedostoihin, haetaan uusia tiedostoja nimen perusteella, luodaan ja poistetaan tiedostoja jne.

VFS-kutsut on kuvattu melkein suoraan MIG-kutsuiksi. Erot johtuvat pääosin erilaisista tiedonvälitysperiaatteista. Esimerkkinä voi mainita vaikkapa VFS:n kutsun `rdwr`, tiedostosta luku tai kirjoitus, joka on MIG-rutiineissa korvattu kahdella eri kutsulla. Tämä johtuu siitä, että MIG-kutsuissa tietyn argumentin välittämän tiedon suunta on ennalta määritelty, eikä se voi riippua jonkin muun argumentin arvosta.

Kun tiedostojärjestelmä, kuten UFS:n tapauksessa levypartitio, NFS:n tapauksessa UDP-osoite tai PFS:ssä Machin portti, kiinnitetään `mount` -systeemikutsulla UNIX-palvelimeen, annetaan `mount`-kutsussa mukana tämä kahva kyseiseen tiedostojärjestelmäpalvelimeen. PFS:n tapauksessa `mount`-kutsulle annetaan portti, johon kyseistä PFS:ää palvelevalla kehyksellä on vastaanotto-oikeus. UNIX-palvelin saa kaikki tähän tiedostojärjestelmään liittyvät palvelut tätä porttia tai sen avulla hankittuja muita portteja käyttäen.

11 Johtopäätökset

Alussa tarkasteltiin UNIXin ja joidenkin muiden yleisten käyttöjärjestelmien tiedostojärjestelmätoteutuksia. UNIXia käsiteltiin tarkemmin ja tyypillisten käyttöesimerkkien avulla näytettiin, että sen tiedostojärjestelmäliityntä on epäjohdonmukainen ja kömpelö.

Hajautettuja tiedostojärjestelmiä tarkasteltiin tutkimalla, minkälaisia ratkaisuja niiden toteutusten pohjaksi oli valittu. Havaintona oli, että uudemmat huolella suunnitellut tiedostojärjestelmät toimivat kohtuullisesti, vaikkakin UNIX-semantiikan toteuttamisessa on puutteita. NFS todettiin monella tavalla huonoksi ja epäluotettavaksi.

Mach esiteltiin mahdollisena pohjana nykyaikaiselle käyttöjärjestelmälle, jonka päälle myös modulaarisen ja haluttaessa hajautetun tiedostojärjestelmän rakentaminen on helpompaa kuin esimerkiksi tavanomaiseen UNIXiin. Todettiin, että Machia käyttäen muu käyttöjärjestelmä on mahdollista eristää tiedostojärjestelmän mahdollisesta virheellisestä toiminnasta. Perinteisissä käyttöjärjestelmissähän tiedostojärjestelmän kaatuminen usein johtaa myös muun käyttöjärjestelmän kaatumiseen.

Mach-ympäristössä ohjelmointia havainnollistettiin esittelemällä yksinkertainen ohjelma, joka tarjoaa oliopohjaisen palvelun. Ohjelmassa on käytetty tärkeimpiä niistä Machin piirteistä, joita tarvitaan esimerkiksi tiedostojärjestelmäpalvelimen rakentamiseksi Mach-ympäristöön.

Machin tiedonsiirto antaa mahdollisuuden hajauttaa tiedostojärjestelmä erillisen palvelimen avulla siten, ettei tiedostojärjestelmän itse tarvitse tietää hajautuksesta. Hajautuspalvelimen on sen sijaan tarpeen tietää, että se välittää tiedostojärjestelmään liittyvää liikennettä.

Lopuksi esiteltiin testausympäristö, jota voi käyttää tiedostojärjestelmien sekä Machin päällä ajettavien käyttöjärjestelmien turvalliseen testaukseen. Vaikka testattava käyttöjärjestelmä tai tiedostojärjestelmä kaatuisi, pohjalla ajettavat Mach-ydin ja normaali käyttöjärjestelmä voivat edelleen toimia häiriöttä. Testattavaa tiedosto- tai käyttöjärjestelmää voidaan lisäksi hallita kaikilla normaaleilla ohjelmankehitysvälineillä, kuten virheidenetsintäohjelmilla.

Varsinaisena tavoitteena ollut tiedostojärjestelmäpalvelin toteutettiin tässä vaiheessa tarjoamaan vain UNIXissa muutenkin olevat tiedostopalvelut Machin porttien välityksellä. Tällä mah-

dollistettiin tiedostojärjestelmien jakaminen yhtäaikaa ajettavien käyttöjärjestelmäpalvelimien kesken. Perusratkaisu havaittiin toimivaksi ja monipuolisempien myöhemmin käyttöön otettavien tiedostojärjestelmäpalvelimien käyttö saman liitynnän avulla on helppoa.

Testausympäristö on osoittautunut hyödylliseksi mm. OSF/1:n testauksessa ja on päivittäisessä käytössä TKK:n Laskentakeskuksessa. Sitä kehitetään edelleen, ja siihen kuuluva porttitiedostojärjestelmä on tarkoitus sovittaa OSF/1:een ja muihin Machin päällä toimiviin käyttöjärjestelmiin.

A VFS-kutsut

```
int
pfs_open(vpp,flag,cred)
struct vnode     **vpp;
int              flag;
struct ucred     *cred;

int
pfs_close(vp,flag,cred)
struct vnode     *vp;
int              flag;
struct ucred     *cred;

int
pfs_rdwr(vp,uiop,rw,ioflag,cred)
struct vnode     *vp;
struct uio       *uiop;
enum uio_rw      rw;
int              ioflag;
struct ucred     *cred;

int
pfs_rwvp(vp,uio,rw,cred,flags,isfile)
struct vnode     *vp;
struct uio       *uio;
enum uio_rw      rw;
struct ucred     *cred;
int              isfile;

int
pfs_ioctl(vp,com,data,flag,cred)
struct vnode     *vp;
int              com;
caddr_t          data;
int              flag;
struct ucred     *cred;

int
pfs_select(vp,which,cred)
struct vnode     *vp;
int              which;
struct ucred     *cred;

int
pfs_getattr(vp,vap,cred)
struct vnode     *vp;
struct vattr     *vap;
struct ucred     *cred;

int
pfs_setattr(vp,vap,cred)
struct vnode     *vp;
struct vattr     *vap;
struct ucred     *cred;

int
pfs_access(vp,mode,cred)
struct vnode     *vp;
int              mode;
struct ucred     *cred;

int
```

```
pfs_readlink(vp,uiop,cred)
struct vnode      *vp;
struct uio        *uiop;
struct ucred      *cred;

int
pfs_fsync(vp,cred)
struct vnode      *vp;
struct ucred      *cred;

int
pfs_inactive(vp,cred)
struct vnode      *vp;
struct ucred      *crwed;

int
pfs_lookup(dvp,nm,vpp,cred)
struct vnode      *dvp;
char              *nm;
struct vnode      **vpp;
struct ucred      *cred;

int
pfs_create(dvp,nm,vap,exclusive,mode,vpp,cred)
struct vnode      *dvp;
char              *nm;
struct vattr      *vap;
enum vcexcl       exclusive;
int               mode;
struct vnode      **vpp;
struct ucred      *cred;

int
pfs_remove(vp,nm,cred)
struct vnode      *vp;
char              *nm;
struct ucred      *cred;

int
pfs_link(vp,tdvp,tnm,cred)
struct vnode      *vp;
struct vnode      *tdvp;
char              *tnm;
struct ucred      *cred;

int
pfs_rename(sdvp,snm,tdvp,tnm,cred)
struct vnode      *sdvp;
char              *snm;
struct vnode      *tdvp;
char              *tnm;
struct ucred      *cred;

int
pfs_mkdir(dvp,nm,vap,vpp,cred)
struct vnode      *dvp;
char              *nm;
struct vattr      *vap;
struct vnode      **vpp;
struct ucred      *cred;

int
pfs_rmdir(vp,nm,cred)
struct vnode      *vp;
char              *nm;
struct ucred      *cred;
```

```
int
pfs_readdir(vp,uiop,cred)
struct vnode      *vp;
struct uio        *uiop;
struct ucred      *cred;

int
pfs_symlink(dvp,lnm,vap,tnm,cred)
struct vnode      *dvp;
char              *lnm;
struct vattr      *vap;
char              *tnm;
struct ucred      *cred;

int
pfs_bread(vp,lbn,bpp,sizep)
struct vnode      *vp;
daddr_t           lbn;
struct buf        **bpp;
long              *sizep;

int
pfs_brelse(vp,bp)
struct vnode      *vp;
 struct buf       *bp;

int
pfs_bmap(vp,bn,vpp,bnp)
struct vnode      *vp;
daddr_t           bn;
struct vnode      **vpp;
daddr_t           *bnp;

int
pfs_lockctl(vp,ld,cmd,cred)
struct vnode      *vp;
struct flock      *ld;
int               cmd;
struct ucred      *cred;

pfs_fid(vp,fidpp)

struct vnode      *vp;
struct fid        **fidpp;

int
pfs_freefid(vp,fidp)
struct vnode      *vp;
struct fid        *fidp;

int
pfs_page_read(vp,buffer,size,offset,cred)
struct vnode      *vp;
caddr_t           buffer;
int               size;
vm_offset_t       offset;
struct ucred      *cred;

int
pfs_page_write(vp,buffer,size,offset,cred,init)

struct vnode      *vp;
caddr_t           buffer;
int               size;
vm_offset_t       offset;
```

```
struct ucred      *cred;
boolean_t         init;

int
pfs_nlinks(vp,l,cred)
struct vnode      *vp;
int               *l;
struct ucred      *cred;
```

B tserver.c

```
/* This program is based on an example program presented by
   Richard P. Draves at Usenix Mach Symposium at Monterey, California
*/

#include <stdio.h>
#include <mach.h>
#include <cthreads.h>
#include <mach/message.h>
#include <mach/notify.h>
#include <mach_error.h>
#include <mig_errors.h>

#include "tserver.h"
#include "object.h"

#include "sym.h"

extern mach_port_t      new_port();
extern mach_port_t      make_wref();

/* port set to hold all object ports */
static mach_port_t object_ports;

/* an upper bound on messages that we handle */
#define MAX_MSG_SIZE    512

static any_t server_thread();
static boolean_t object_demux();

extern boolean_t tserver_server();
extern boolean_t object_server();
extern char     *strsave();
static struct mutex nservers_lock = MUTEX_INITIALIZER;
static int      max_server_threads = 1;
static int      max_server_threads_idle = 1;
static int      nservers = 0;
static int      nservers_busy = 0;
static int      nservers_idle = 0;
static mach_port_t      kill_port;

int     vflag;

static void
usage()
{
  fprintf(stderr,"usage: tserver [-v]\n");
  exit(1);
}

symtable_t      root;

extern char     *optarg;
extern int      optind;

main(argc, argv)
    int argc;
    char *argv[];
{
    int c,i,num_threads;
    mach_port_t service;

    while ((c = getopt(argc,argv,"vt:")) != EOF) {
```

```
      switch (c) {
      case 'v':
        vflag++;
        break;
      case 't':
        max_server_threads = atoi(optarg);
        max_server_threads_idle = max_server_threads/2;
        break;
      default:
        usage();
      }
    }
    /* Allocate our service port and check it into the name service. */
    service = new_port();
    if (cksys(netname_check_in(name_server_port, "Tserver",
                               mach_task_self(), service),
              "tserver: netname_check_in") != KERN_SUCCESS)
      exit(1);
    /* Allocate a port set and create threads to receive from it. */
    (void) mach_port_allocate(mach_task_self(),
                              MACH_PORT_RIGHT_PORT_SET, &object_ports);
   /* Allocate kill port and add it to the port set. */
    kill_port = make_wref(new_port());
    if (cksys(mach_port_move_member(mach_task_self(), kill_port, object_ports),
              "tserver: mach_port_move_member") != KERN_SUCCESS)
      exit(1);
    root = sym_init();
    cthread_detach(cthread_fork(server_thread, (any_t) object_ports));
    /* We service messages sent to the service port. */
    if (cksys(mach_msg_server(tserver_server, MAX_MSG_SIZE, service),
              "tserver: mach_msg_server") != KERN_SUCCESS)
      return 1;
    return 0;
}

static any_t
server_thread(arg)

any_t arg;

{
    mach_port_t pset = (mach_port_t) arg;

    mutex_lock(&nservers_lock);
    nservers++;
    vflag && printf("tserver: nservers=%3d\n",nservers);
    mutex_unlock(&nservers_lock);
    if (cksys(mach_msg_server(object_demux,
                              MAX_MSG_SIZE, pset),
              "tserver: mach_msg_server") != KERN_SUCCESS)
      exit(1);
    return 0;
}

typedef struct object {
  struct mutex o_lock;          /* lock for the object */
  struct condition o_wait;      /* wait for sequence number */
  unsigned int o_refs;          /* internal refs */

  mach_port_t o_port;           /* port representing the object */
  mach_port_mscount_t o_mscount; /* make-send count for the port */
  mach_port_seqno_t o_seqno;    /* sequence number for the port */
  int o_value;                  /* the object's value */
  char  *name;
} *object_t;
```

```
/*
 *      To simplify things, we actually only manage one object.
 *      The global lock protects symbol table.
 *      Each object has its own lock for its fields.
 *      The global lock must be taken before an object lock.
 */

static struct mutex global_lock = MUTEX_INITIALIZER;

static void do_no_senders();
static mach_port_t malloc_object_and_port();

static object_t
convert_port_to_object(port)

mach_port_t port;

{
    object_t object;

    /* convert a port to an object, returning a reference */
    object = (object_t) port;

    mutex_lock(&object->o_lock);
    object->o_refs++;
    mutex_unlock(&object->o_lock);

    return object;
}

static void
release_object(object)
object_t object;
{
  unsigned int refs;

  /* release a reference for the object */

  mutex_lock(&object->o_lock);
  refs = --object->o_refs;
  vflag &&
    printf("tserver: no senders, release object %s refs=%d\n",object->name,refs);
  mutex_unlock(&object->o_lock);

  if (refs != 0)
    return;

  (void) mach_port_mod_refs(mach_task_self(), object->o_port,
                            MACH_PORT_RIGHT_RECEIVE, -1);
  free(object->name);
  free((char *) object);
}

static boolean_t
object_demux(request, reply)
mach_msg_header_t *request, *reply;
{
  object_t object;

  /* We try to keep enough object threads running all the time */
  mutex_lock(&nservers_lock);

  if (request->msgh_local_port == kill_port) {
    /* We have to kill ourself */
    nservers--;
    vflag &&
```

```
    printf("tserver: nservers=%3d nservers_busy=%3d\n",nservers,nservers_busy);
  mutex_unlock(&nservers_lock);
  cthread_exit(0);
}
nservers_busy++;
vflag &&
  printf("tserver: nservers=%3d nservers_busy=%3d\n",nservers,nservers_busy);
if (nservers_busy >= nservers && nservers < max_server_threads)
  cthread_detach(cthread_fork(server_thread, (any_t) object_ports));
mutex_unlock(&nservers_lock);

/* translate the port into an object, acquiring a ref for the object */

object = convert_port_to_object(request->msgh_local_port);

/*
 *    We wait for any previous requests to get this far.
 *    In particular, we can't process a no-senders notification
 *    for a port while some previous requests on the port haven't
 *    yet translated the port into an object and taken a ref for
 *    the object.
 */

mutex_lock(&object->o_lock);
while (object->o_seqno < request->msgh_seqno)
  condition_wait(&object->o_wait, &object->o_lock);
object->o_seqno++;
mutex_unlock(&object->o_lock);
condition_broadcast(&object->o_wait);

/*
 *    If this message was sent to a send-once right,
 *    then it must be a legitimate no-more-senders notification.
 *    (We give send-once rights only to the kernel.)
 *    Otherwise use object_server, passing the object.
 */

if (MACH_MSGH_BITS_LOCAL(request->msgh_bits) ==
    MACH_MSG_TYPE_PORT_SEND_ONCE) {
  mach_no_senders_notification_t *n =
    (mach_no_senders_notification_t *) request;
  mig_reply_header_t *r = (mig_reply_header_t *) reply;

  /* handle the notification */

  do_no_senders(object, n->not_count);

  /* tell mach_msg_server not to send a reply.
     we need this because we didn't use notify_server() */

  r->Head.msgh_bits = 0;
  r->Head.msgh_remote_port = MACH_PORT_NULL;
  r->RetCode = KERN_SUCCESS;
} else {
  /* this assignment is only really necessary
     if the object address and the port name might be different */
  request->msgh_local_port = (mach_port_t) object;
  (void) object_server(request, reply);
}

/* release our reference for the object */

release_object(object);
{
  int nservers_idle;
  mutex_lock(&nservers_lock);
```

```
    nservers_busy--;
    nservers_idle = nservers - nservers_busy;
    if (nservers_idle > 2 && nservers_idle > max_server_threads_idle) {
      vflag && printf("tserver: Sending KILL message\n",nservers);
      empty_message(kill_port);
    }
    mutex_unlock(&nservers_lock);
  }
  return TRUE;
}

kern_return_t
do_object_change(port, value)
mach_port_t port;
int value;
{
  object_t object = (object_t) port;

  mutex_lock(&object->o_lock);
  object->o_value = value;
  mutex_unlock(&object->o_lock);
  if (value / 100 == 99)
    port_sleep((value%100)*1000);

  return KERN_SUCCESS;
}

kern_return_t
do_object_query(port, valuep)
mach_port_t port;
int *valuep;
{
  object_t object = (object_t) port;
  int value;

  mutex_lock(&object->o_lock);
  value = object->o_value;
  mutex_unlock(&object->o_lock);

  *valuep = value;
  return KERN_SUCCESS;
}

remove_name(name)

o_name_t        name;

{
  sym_delete(root,name);
}

static void
do_no_senders(object, mscount)
object_t object;
mach_port_mscount_t mscount;
{
  /* the locking hierarchy requires that we take
     the global lock first, then the object lock */

  mutex_lock(&global_lock);
  mutex_lock(&object->o_lock);
  if (mscount == object->o_mscount) {
    int final;
    /*
     *  It is safe for us to shut down the object,
     *  so we remove the global reference to the object.
```

```
     *  The object will get deallocated when release_object
     *  removes the last reference for it (probably our ref).
     */
    final = object->o_value == 999999;
    remove_name(object->name);
    mutex_unlock(&global_lock);
    object->o_refs--;
    vflag && printf("tserver: no senders, removing object %s\n",object->name);
    if (final) {
      printf("tserver: exititing\n");
      exit(0);
    }
  } else {
    mach_port_t previous;
    kern_return_t kr;

    mutex_unlock(&global_lock);
    vflag && printf("tserver: no senders, still references %s\n",object->name);

    /*
     *  This means a do_tserver_lookup slipped in and
     *  created another send right for the object,
     *  so we shouldn't destroy it yet.  We request
     *  another no-senders notification instead.
     */

    kr = mach_port_request_notification(mach_task_self(),
                                        object->o_port,
                                        MACH_NOTIFY_NO_SENDERS, object->o_mscount,
                                        object->o_port, MACH_MSG_TYPE_MAKE_SEND_ONCE,
                                        &previous);
    if ((kr != KERN_SUCCESS) || (previous != MACH_PORT_NULL)) {
      cksys(kr,"tserver: mach_port_request_notification: %s\n");
      exit(1);
    }
  }
  mutex_unlock(&object->o_lock);
}

kern_return_t
do_tserver_lookup_byname(server, name, portp)
mach_port_t     server;
o_name_t        name;
mach_port_t     *portp;
{
  mach_port_t port;
  object_t      *r,op;

  mutex_lock(&global_lock);

  r = (object_t*)sym_lookup(root,name,1);

  if ((op = *r) == NULL) {
    mach_port_t previous;
    kern_return_t kr;

    /* create a new object and port for it */
    vflag && printf("tserver: creating new object %s\n",name);

    port = malloc_object_and_port(sizeof *op);
    op = *r = (object_t) port;
    op->name = strsave(name);

    /* put the object port into the port set */

    if (cksys(mach_port_move_member(mach_task_self(), port, object_ports),
```

```
            "tserver: mach_port_move_member") != KERN_SUCCESS)
        exit(1);
      /* request the initial no-senders notification */
      kr = mach_port_request_notification(mach_task_self(), port,
                                          MACH_NOTIFY_NO_SENDERS, 1,
                                          port, MACH_MSG_TYPE_MAKE_SEND_ONCE,
                                          &previous);
      if ((kr != KERN_SUCCESS) || (previous != MACH_PORT_NULL)) {
        cksys(kr,"tserver: mach_port_request_notification");
        exit(1);
      }
      mutex_init(&op->o_lock);
      condition_init(&op->o_wait);
      op->o_refs = 1;      /* the global ref */
      op->o_port = port;
      op->o_mscount = 0;
      op->o_seqno = 0;
      op->o_value = 0;
    }
    /* return a send right for the object */

    mutex_lock(&op->o_lock);
    mutex_unlock(&global_lock);
    port = op->o_port;
    op->o_mscount++;
    mutex_unlock(&op->o_lock);

    *portp = port;
    return KERN_SUCCESS;
}

extern char *malloc();

/* allocate a structure and a port together, so that
   the structure's address and the port's name are the same */

static mach_port_t
malloc_object_and_port(size)
unsigned int size;

{
    char *badobjects = NULL;
    char *object;
    kern_return_t kr;

    for (;;) {
      object = malloc(size);
      if (object == NULL)
        quit(1, "tserver: malloc failed\n");

      /* try to allocate a port with this name */

      kr = mach_port_allocate_name(mach_task_self(),
                                   MACH_PORT_RIGHT_RECEIVE,
                                   (mach_port_t) object);
      if (kr == KERN_SUCCESS)
        break;
      if (kr != KERN_NAME_EXISTS) {
        cksys(kr,"tserver: mach_port_allocate_name");
        exit(1);
      }
      /* save this structure and try again */
      * (char **) object = badobjects;
      badobjects = object;
    }
```

```
  /* free the unused objects */

  while (badobjects != NULL) {
    char *next = * (char **) badobjects;
    free(badobjects);
    badobjects = next;
  }

  return (mach_port_t) object;
}
```

C sym.c

```
#include <stdio.h>
#include "sym.h"

extern char     *malloc();

typedef struct symtable symtable_rec;

struct symtable {
  symtable_t    next;
  symtable_t    prev;
  char          *name;
  char          *value;
};

struct symtable_hdr {
  symtable_t    next;
  symtable_t    prev;
};

symtable_t
sym_init()

{
  struct symtable_hdr   *p;
  p = (struct symtable_hdr*)malloc(sizeof(*p));
  if (!p) quit(1,"Cannot malloc\n");
  p->next = p->prev = ((symtable_t)p);
  return (symtable_t)p;
}

sym_delete(root,name)

symtable_t      root;
char            *name;

{
  symtable_t    pp,p;

  for(pp = root; (p = pp->next) != root; pp = p) {
    if (strcmp(p->name,name) == 0) {
      free(p->name);
      p->next->prev = p->prev;
      p->prev->next = p->next;
      free(p);
      return 1;
    }
  }
  return 0;
}

char **
sym_lookup(root,name,create)

symtable_t      root;
char            *name;
int             create;

{
  symtable_t    pp,p;

  for(pp = root; (p = pp->next) != root; pp = p) {
    if (strcmp(p->name,name) == 0)
```

```
      return &(p->value);
  }
  if (create) {
    if (!(p = (symtable_t)malloc(sizeof(*p)))) quit(1,"Cannot malloc\n");
    if (!(p->name = malloc(strlen(name)+1))) quit(1,"Cannot malloc\n");
    strcpy(p->name,name);
    p->next = pp->next;
    pp->next = p;
    p->prev = pp;
    p->value = 0;
    return &(p->value);
  } else
    return 0;
}

sym_iterate(root,funptr)

symtable_t      root;
int             (*funptr)();

{
  symtable_t    pp,p;

  for(pp = root; (p = pp->next) != root; pp = p) {
    funptr(p->name,p->value);
  }
}
```

Viitteet

[CMU91] Open Software Foundation and Carnegie Mellon University. *Mach 3 Kernel Interface*, May 1991.

[DA92] Randall W. Dean and Francois Armand. Data movement in kernelized systems. Anonymous FTP at `mach.cs.cmu.edu:/usr/mach/public/doc/mach3/datamovement.{ps,doc}`, School of Computer Science, Carnegie Mellon University, March 1992.

[Dra91] Richard P. Draves, Jr. A revised IPC interface, 1991.

[LMKQ89] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

[Loe91a] Open Software Foundation and Carnegie Mellon University. *Mach 3 Kernel Principles*, March 1991.

[Loe91b] Open Software Foundation and Carnegie Mellon University. *Server Writer's Guide*, May 1991.

[Loe91c] Open Software Foundation and Carnegie Mellon University. *Server Writer's Interface*, May 1991.

[McK84] Marshall K. McKusick. A fast file system for Unix. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[Mul85] Sape J. Mullender. A distributed file service based on optimistic concurrency control. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 51–62. ACM, Dec 1985.

[NFS89] NFS: Network file system protocol specification. RFC-1094, RFCs are available at your nearest network archive, 1989.

[OCD+88] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *IEEE Computer*, pages 23–36, Feb 1988.

[RFC768] User datagram protocol. RFCs are available at your nearest network archive.

[RT78] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 57(6):1905–1929, July-August 1978.

[SGN85] Michael D. Schroeder, David K. Gifford, and Roger M. Needham. A caching file system for a programmer's workstation. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 25–34. ACM, Dec 1985.

[SPG91] Abraham Silberschatz, James L. Peterson, and Peter B. Galvin. *Operating System Concepts*. Addison-Wesley, third edition, 1991.

[VMS88] *VMS General User's Manual*. Digital Equipment Corporation, 1988.

Teko Lalle
27.8.1992